

1introduction to compiler.pdf

3Lexical_Analysis.pdf

4syntax analysis(parsing).pdf

5syntax directed translation.pdf

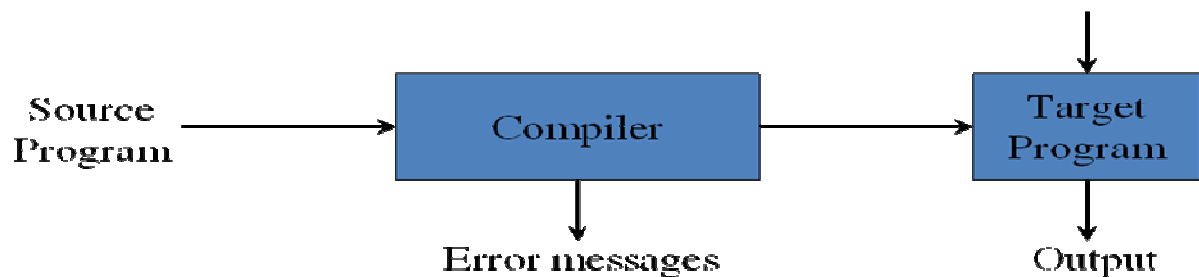
6 Type Checking.pdf

8intermediate code generation.pdf

9code generation and optimization.pdf

What are compilers

- To most people, a compiler is a "black box program" which takes source code written in some high-level language, such as FORTRAN, BASIC, Pascal, or C, and translates (compiles) it into a language the computer can understand and execute. Compilers take source code and transform it into an equivalent code in a target language.

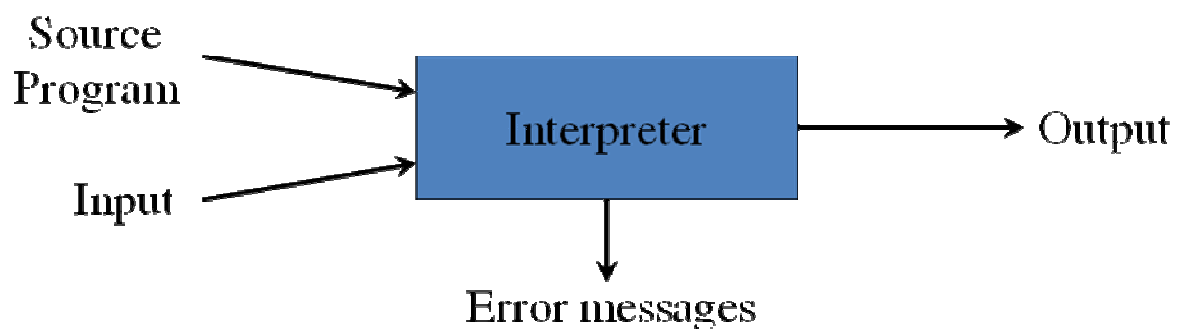


Cousins of the Compiler

Pre-Processors: The pre-processors are those programs which perform a pre-compilation of the source program to expand any macro definitions.

Loader and Linkers: If the target program is machine code, loaders are used to load the target code into memory for execution. Linkers are used to link target program with the libraries.

Interpreter: Interpreters perform compilation, loading and execution in lock –steps



JIT Compilers: Just-in-time(JIT) compilers perform complete compilation followed immediately by loading and execution. JIT compilers represent a hybrid approach, with translation occurring continuously, as with interpreters, but with caching of translated code to minimize performance degradation

Compilers vs. Interpreters

- Computers cannot understand English words and grammar. Even the highly structured words and sentences of programming languages must be translated before a computer can understand them.
- The compiler or interpreter must look up each "word" of our programming language in a kind of dictionary (or lexicon) and, in a series of steps, translate it into machine code. Each word initiates a separate logical task.
- An interpreter translates one line of source code at a time into machine code, and then executes it. Debugging and testing is relatively fast and easy in interpreted languages, since the entire program doesn't have to be reprocessed each time a change is made. E.g. BASIC, COBOL, MySQL
- Interpreted programs run much slower than compiled programs, because they must be translated each time they are run. Programmers often test and debug their programs using an interpreter and then compile them for production use.

How Compilers Work

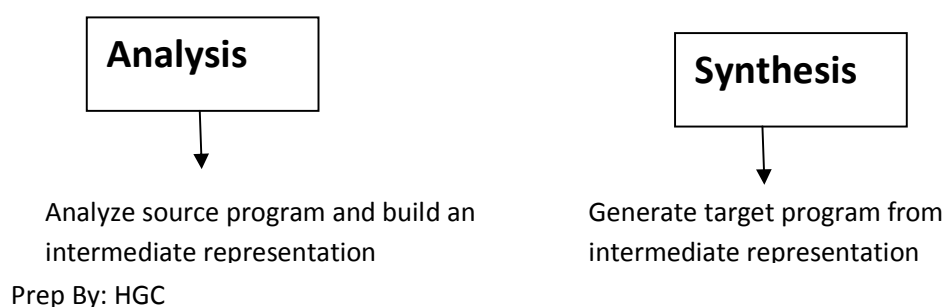
Most compilers convert programs in three steps. Each step is called a pass. A particular compiler may have one program per pass, or may combine two or three steps in a single program. For a very complex language, a step may be so difficult to perform that it is broken up into many smaller steps. Regardless of how many passes or programs are required, the compiler performs only three main functions:

1. Lexical analysis: The stream of characters forming the source program are scanned linearly to produce a stream of logical elements called *tokens*.
2. Syntax analysis: The stream of tokens are grouped into hierarchical collections called syntax groupings.
3. Code generation. Generating code in the target language.

During each pass of the compiler, the source code moves closer to becoming virtual machine language (or whatever language the compiler is designed to generate).

Phases of Compilers

For compilation of any source program written in some source language, the compiler passes from Analysis and Synthesis phases

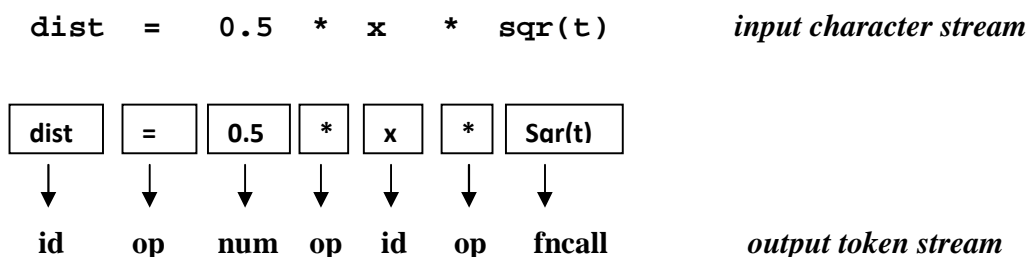


Analysis of source program includes: Lexical analysis, Syntax analysis and Semantic analysis

Lexical Analysis

In the first pass of the compiler, the source code is passed through a lexical analyzer, which converts the source code to a set of tokens. A token generally representing some keyword in the language. A compiler has a unique number for each keyword (i.e. IF, WHILE, END), and each arithmetic or logical operator (i.e. +, -, *, AND, OR, etc.). Numbers are represented by a token which indicates that what follows it should be interpreted as a number.

The other important task of the lexical analyzer is to build a symbol table. This is a table of all the identifiers (variable names, procedures, and constants) used in the program. When an identifier is first recognized by the analyzer, it is inserted into the symbol table, along with information about its type, where it is to be stored, and so forth. This information is used in subsequent passes of the compiler. E.g.



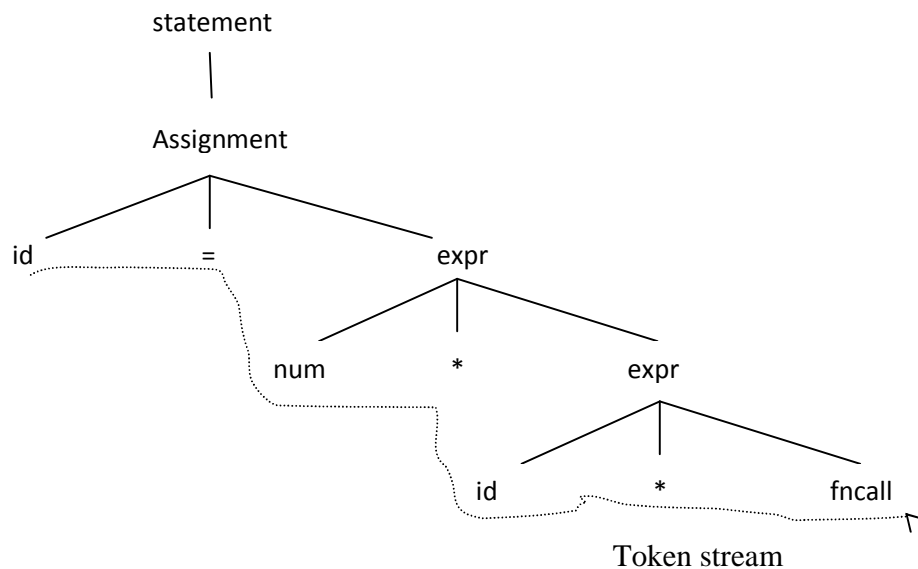
id: identifier, **op:** operator, **num:** number, **fncall:** function call

Syntax Analysis

After the lexical analyzer translates a program into tokens of keywords, variables, constants, symbols and logical operators, the compiler makes its next pass. To describe what happens during this function, recall the concept of grammars.

Grammars. Like any language, programming languages have a set of rules governing the structure of the program. Each different computer language has its own grammar which makes it unique. Some grammars are complex and others are relatively easy. The programmer must observe all the structural rules of a language to make logical sense to the computer. The next step of the compiling process, parsing, checks to be sure all the rules were followed.

Parsing. The parsing routines of a compiler check to see that the program is written correctly (according to the language rules described by grammars). The parser reads in the tokens generated by the lexical analyzer and compares them to the set grammar of the programming language. If the program follows the rules of the language, then it is syntactically correct. When the parser encounters a mistake, it issues a warning or error message and tries to continue.



Example of parse tree construction from token stream

Some parsers try to correct a faulty program, others do not. When the parser reaches the end of the token stream, it will tell the compiler that either the program is grammatically correct and compiling can continue or the program contains too many errors and compiling must be aborted. If the program is grammatically correct, the parser will call for semantic routines.

Semantic Analysis

The semantic routines of a compiler perform two tasks: checking to make sure that each series of tokens will be understood by the computer when it is fully translated to machine code, and converting the series of tokens one step closer to machine code.

- The first task takes a series of tokens, called a *production*, and checks it to see if it makes sense. For example, a production may be correct as far as the parser is concerned, but the semantic routines check whether the variables have been declared, and are of the right type, etc.
- If the production makes sense, the semantic routine reduces the production for the next phase of compilation, code generation. Most of the code for the compiler lies here in the semantic routines and thus takes up a majority of the compilation time.

Thus, two major routines comprise syntax analysis: the parsing routine and the semantic routine. The parser checks for the correct order of the tokens and then calls the semantic routines to check whether the series of tokens (a production) will make sense to the computer. The semantic routine then reduces the production another step toward complete translation to machine code.

Synthesis : concerns issues involving generating code in the target language. It usually consists of the following phases.

- **Intermediate code Generation**
- **Code optimization**
- **Final Code generation**

An intermediate language is often used by many compilers for analyzing and optimizing the source program. Intermediate language should have two important properties:

It should be simple and easy to produce and

It should be easy to translate target program.

Example:

```
temp1 = int2float(sqr(t));  
temp2 = g*temp1;  
temp3 = 0.5*temp2,  
dist = temp3
```

The code generation process determines how fast the code will run and how large it will be. The first part of code generation involves optimization, and the second involves final target code generation.

Code Optimization. In this step, the compiler tries to make the intermediate code generated by the semantic routines more efficient. This process can be very slow and may not be able to improve the code much. Because of this, many compilers don't include optimizers, and, if they do, they look only for areas that are easy to optimize.

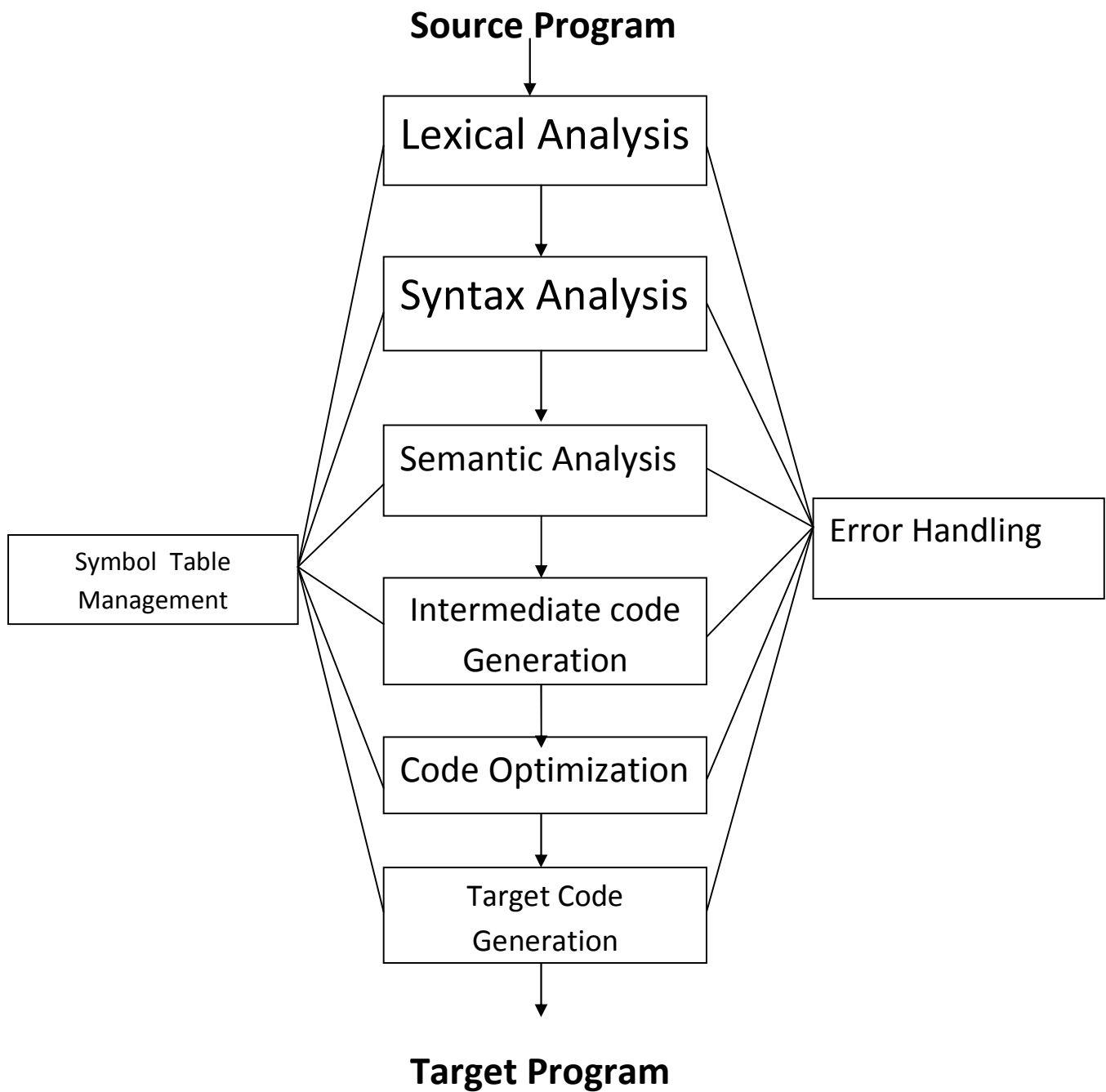
- Detection of redundant function call
- Detection of loop invariants
- Common sub expression compilation
- Dead-code detection and elimination

Final Code Generation. This process takes the intermediate code produced by the optimizer (or semantic routines if the compiler has no optimizer) and generates final code in target language. It is this part of the compilation phase that is machine dependent. Each type of computer has an operating system that processes virtual machine code differently; therefore, the code generator must be different for each type of computer.

If the program is free from syntactical errors, code generation should take place without any problem. When the code generator is finished, the code produced will be in target code.

*It is in a format (an .OBJ file in C compiler) that is ready to go to a linker, which will create an executable *.EXE or *.COM etc file from the machine code the compiler has generated.*

Summary: Phases of a Compiler



The Symbol table: A symbol table stores information about keywords and tokens found during lexical analysis. The symbol table is consulted in almost all phases of the compiler.

For example:

```
insert("dist",id); // inserts a symbol table entry associating the string "dist" with token type : id
lookup("dist"); // an occurrence of the string "dist" can be looked up in the symbol table. If
found, a reference to its token type is returned , else lookup return 0.
```

Error handling:

- Errors may be encountered in any phases of compiler.
- Objective of error handling is to go as far as possible in the compilation whenever an error is encountered.

For example:

- Handling missing symbols during lexical analysis by inserting symbols.
- Automatic type conversion during semantic analysis.

[Lexical Analysis]

Compiler Design and Construction (CSc 352)

Compiled By

Bikash Balami

Central Department of Computer Science and Information Technology (CDCSIT)

Tribhuvan University, Kirtipur

Kathmandu, Nepal

Lexical Analysis

This is the initial part of reading and analyzing the program text. The text is read and divided into tokens, each of which corresponds to a symbol in a programming language, e.g. a variable name, keyword or number etc. So a lexical analyzer or lexer, will as its input take a string of individual letters and divide this string into tokens. It will discard comments and white-space (i.e. blanks and newlines).

Overview of Lexical Analysis

A lexical analyzer, also called a scanner, typically has the following functionality and characteristics.

- Its primary function is to convert from a sequence of characters into a sequence of tokens. This means less work for subsequent phases of the compiler.
- The scanner must Identify and Categorize specific character sequences into tokens. It must know whether every two adjacent characters in the file belong together in the same token, or whether the second character must be in a different token.
- Most lexical analyzers discard comments & whitespace. In most languages these characters serve to separate tokens from each other.
- Handle lexical errors (illegal characters, malformed tokens) by reporting them intelligibly to the user.
- Efficiency is crucial; a scanner may perform elaborate input buffering
- Token categories can be (precisely, formally) specified using regular expressions, e.g.
- IDENTIFIER = [a-zA-Z][a-zA-Z0-9]*
- Lexical Analyzers can be written by hand, or implemented automatically using finite automata.

Role of Lexical Analyzer

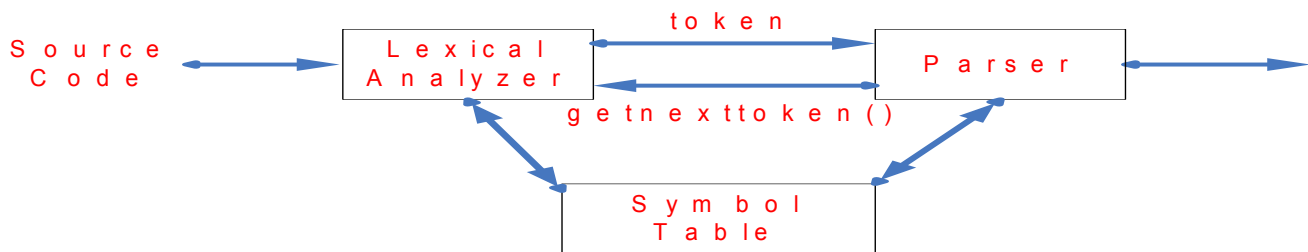


Figure:- Interaction of lexical analyzer with parser

The lexical analyzer works in lock step with the parser. The parser requests the lexical analyzer for the next token whenever it requires one using `getnexttoken()`. Lexical analyzer may also perform other auxiliary operations like removing redundant white spaces, removing token separators (like semicolon ;) etc. Some other operations performed by lexer, includes removal of comments, providing line number to the parser for error reporting. The Function of a lexical Analyzer is to read the input stream representing the Source program, one character at a time and to translate it into valid tokens.

Issues in Lexical Analysis

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing.

- 1) Simpler design is the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.
- 2) Compiler efficiency is improved.
- 3) Compiler portability is enhanced.

Tokens, Patterns, Lexemes

In compiler, a *token* is a single word of source code input. Tokens are the separately identifiable block with collective meaning. When a string representing a program is broken into sequence of substrings, such that each substring represents a constant, identifier, operator, keyword etc of the language, these substrings are called the tokens of the Language. They are the building block of the programming language. E.g. if, else, identifiers etc.

Lexemes are the actual string matched as token. They are the specific characters that make up of a token. For example ***abc*** and ***123***. A token can represent more than one lexeme. i.e. token ***intnum*** can represent lexemes ***123***, ***244***, ***4545*** etc.

Patterns are rules describing the set of lexemes belonging to a token. This is usually the regular expression. E.g. ***intnum*** token can be defined as $[0-9][0-9]^*$.

Attribute of tokens

When a token represents more than one lexeme, lexical analyzer must provide additional information about the particular lexeme. i.e. In case of more than one lexeme for a token, we need to put extra information about the token. This additional information is called the *attribute* of the token. For e.g. token ***id*** matched ***var1***, ***var2*** both, here in this case lexical analyzer must be able to represent ***var1***, and ***var2*** as different identifiers.

Example: take statement, ***area = 3.1416 * r * r***

1. `getnexttoken()` returns (id, attr) where *attr* is pointer to ***area*** to symbol table
2. `getnexttoken()` returns (assign) no attribute is needed, if there is only one assignment operator
3. `getnexttoken()` returns (floatnum, 3.1416) where 3.1416 is the actual value of ***floatnum***

.... So on.

Token type and its attribute uniquely identify a lexeme.

Lexical Errors

Though error at lexical analysis is normally not common, there is possibility of errors. When the error occurs the lexical analyzer must not halt the process. So it can print the error message and continue. Error in this phase is found when there are no matching string found as given by the pattern. Some error recovery techniques includes like deletion of extraneous character, inserting missing character, replacing incorrect character by correct one, transposition of adjacent characters etc. Lexical error recovery is normally expensive process. For e.g. finding the number of transformation that would make the correct tokens.

Implementing Lexical Analyzer (Approaches)

1. Use lexical analyzer generator like flex that produces lexical analyzer from the given specification as regular expression. We do not take care about reading and buffering the input.
2. Write a lexical analyzer in general programming language like C. We need to use the I/O facility of the language for reading and buffering the input.
3. Use the assembly language to write the lexical analyzer. Explicitly manage the reading of input.

The above strategies are in increasing order of difficulty, however efficiency may be achieved and as a matter of fact since we deal with characters in lexical analysis, it is better to take some time to get efficient result.

Look Ahead and Buffering

Most of the time recognizing tokens needs to look ahead several other characters after the matched lexeme before the token match is returned. For e.g. *int* is a keyword in C but *integer* is an identifier so when the scanner reads i, n, t then this time it has to look for other characters to see whether it is just *int* or some other word. In this case at next time we need move back to rescan the input again for the characters that are not used for the lexeme and this is time consuming. To reduce the overhead, and efficiently move back and forth *input buffering* technique is used.

Input Buffering

We will consider look ahead with 2N buffering and using the sentinels.

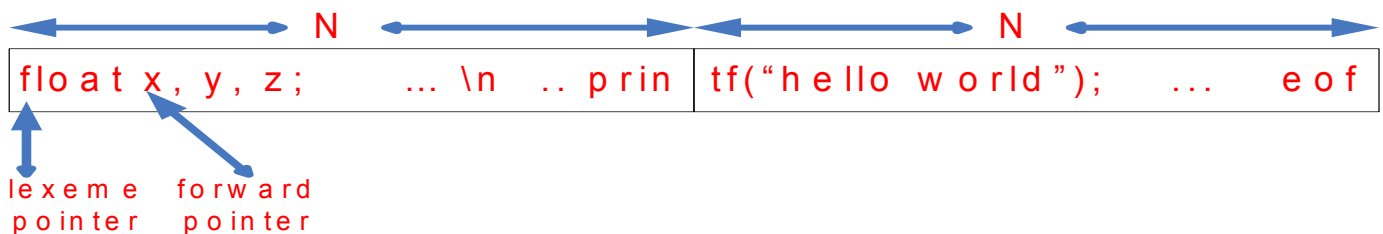


Figure:- An input buffer in two halves

We divide the buffer into two halves with N-characters each. normally N is the number of characters in one disk block like 1024 or 4096. Rather than reading character by character from

Bikash Balami

file we read N input character at once. If there are fewer than N character in input eof marker is placed. There are two pointers (see fig in previous slide). The portion between lexeme pointer and forward pointer is current lexeme. Once the match for pattern is found both the pointers points at same place and forward pointer is moved. This method has limited look ahead so may not work all the time say multi line comment in C. In this approach we must check whether end of one half is reached (two test each time if forward pointer is not at end of halves) or not each time the forward pointer is moved. The number of such tests can be reduced if we place sentinel character at the end of each half.

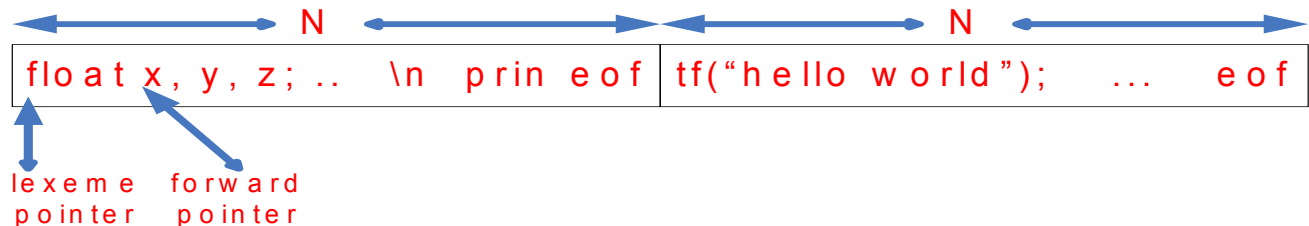


Figure:- Sentinels at end of each buffer half

```

if forward points end of first half
    reload second half
    forward++;
else if forward points end of second half
    reload first half
    forward = start of first half
else
    forward++;

```

Figure:- Code to advance forward pointer(first scheme)

```

forward++;
if forward points eof
    if forward points end of first half
        reload second half
        forward++;
    else if forward points end of second half
        reload first half

```

forward = start of first half

else

terminate lexical analysis

Figure:- Lookahead code with sentinels

Specifications of Tokens

Regular expressions are an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions will serve as names for sets of strings.

Some Definitions

- **Alphabets**
 - An alphabet A is a set of symbols that generate languages. For e.g. $\{0-9\}$ is an alphabet that is used to produce all the non negative integer numbers. $\{0-1\}$ is an alphabet that is used to produce all the binary strings.
- **String**
 - A string is a finite sequence of characters from the alphabet A . Given the alphabet A , $A^2 = A.A$ is set of strings of length 2, similarly A^n is set of strings of length n . The **length** of the string w is denoted by $|w|$ i.e. number of characters (symbols) in w . We also have $A^0 = \{\epsilon\}$, where ϵ is called empty string. $|s|$ denotes the length of the string s .
- **Kleene Closure**
 - Kleene closure of an alphabet A denoted by A^* is set of all strings of any length (0 also) possible from A . Mathematically $A^* = A^0 \cup A^1 \cup A^2 \cup \dots$. For any string, w over alphabet A , $w \in A^*$.
- A **language** L over alphabet A is the set such that $L \subseteq A^*$.
- The string s is called **prefix** of w , if the string s is obtained by removing zero or more trailing characters from w . If s is a proper prefix, then $s \neq w$.
- The string s is called **suffix** of w , if the string s is obtained by deleting zero or more leading characters from w . we say s as proper suffix if $s \neq w$.
- The string s is called **substring** of w if we can obtain s by deleting zero or more leading or trailing characters from w . We say s as proper substring if $s \neq w$.
- Regular Operators

- The following operators are called **regular operators** and the language formed called **regular language**.
 - $\cdot \rightarrow$ Concatenation operator, $R.S = \{rs \mid r \in R \text{ and } s \in S\}$.
 - $*$ \rightarrow Kleene $*$ operator, $A^* = \bigcup_{i=1}^{\infty} A^i$
 - $+/\cup/| \rightarrow$ Choice/union operator, $R \cup S = \{t \mid t \in R \text{ or } t \in S\}$.

Regular Expression (RE)

We use regular expression to describe the tokens of a programming language.

Basis Symbol

- ϵ is a regular expression denoting language $\{\epsilon\}$
- $a \in A$ is a regular expression denoting $\{a\}$

If r and s are regular expressions denoting languages $L_1(r)$ and $L_2(s)$ respectively, then

- $r + s$ is a regular expression denoting $L_1(r) \cup L_2(s)$
- rs is a regular expression denoting $L_1(r) L_2(s)$
- r^* is a regular expression denoting $(L_1(r))^*$
- (r) is a regular expression denoting $L_1(r)$

Examples

- $(1+0)^*0$ is RE that gives the binary strings that are divisible by 2.
- $0^*10^*+0^*110^*$ is RE that gives binary strings having at most 2 1s.
- $(1+0)^*00$ denotes the language of all strings that ends with 00 (binary number multiple of 4)
- $(01)^* + (10)^*$ denotes the set of all strings that describes alternating 1s and 0s

Properties of RE

- $r+s = s+r$ ($+$ is commutative)
- $r+(s+t) = (r+s)+t$; $r(st) = (rs)t$ ($+$ and \cdot are associative)
- $r(s+t) = (rs)+(rt)$; $(r+s)t = (rt)+(st)$ (\cdot distributes over $+$)
- $\epsilon r = r\epsilon$ (ϵ is identity element)
- $r^* = (r+\epsilon)^*$ (relation between $*$ and ϵ)
- $r^{**} = r^*$ ($*$ is idempotent)

Regular Definitions

To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use regular definitions. i.e defining RE giving a name and reusing it as basic symbol to produce another RE, gives the regular definition.

Example

- $d_1 \rightarrow r_1, d_2 \rightarrow r_2, \dots, d_n \rightarrow r_n$, Where each d_i s are distinct name and r_i s are REs over the alphabet $A \cup \{d_1, d_2, \dots, d_{i-1}\}$
- In C the RE for identifiers can be written using the regular definition as
 - $\text{letter} \rightarrow a + b + \dots + z + A + B + \dots + Z.$
 - $\text{digit} \rightarrow 0 + 1 + \dots + 9.$
 - $\text{identifier} \rightarrow (\text{letter} + _)(\text{letter} + \text{digit} + _)^*$

[Note: Remember recursive regular definition may not produce RE, that means

$\text{digits} \rightarrow \text{digits digits} \mid \text{digits}$ is wrong!!!

Recognition of Tokens

A recognizer for a language is a program that takes a string w , and answers “YES” if w is a sentence of that language, otherwise “NO”. The tokens that are specified using RE are recognized by using transition diagram or finite automata (FA). Starting from the start state we follow the transition defined. If the transition leads to the accepting state, then the token is matched and hence the lexeme is returned, otherwise other transition diagrams are tried out until we process all the transition diagram or the failure is detected. Recognizer of tokens takes the language L and the string s as input and try to verify whether $s \in L$ or not. There are two types of Finite Automata.

1. Deterministic Finite Automaton (DFA)
2. Non Deterministic Finite Automaton (NFA)

Deterministic Finite Automaton (DFA)

FA is deterministic, if there is exactly one transition for each (state, input) pair. It is faster recognizer but it make take more spaces. DFA is a five tuple (S, A, S_0, δ, F) where,

$S \rightarrow$ finite set of states

$A \rightarrow$ finite set of input alphabets

$S_0 \rightarrow$ starting state

$\delta \rightarrow$ transition function i.e. $\delta: S \times A \rightarrow S$

$F \rightarrow$ set of final states $F \subseteq S$

Implementing DFA

The following is the algorithm for simulating DFA for recognizing given string. For a given string w , in DFA D , with start state q_0 , the output is “YES”, if D accepts w , otherwise “NO”.

```

DFASim( $D, q_0$ ) {
     $q = q_0$ ;
     $c = \text{getchar}()$ ;
    while ( $c \neq \text{eof}$ )
    {
         $q = \text{move}(q, c)$ ; //this is  $\delta$ function.
         $c = \text{getchar}()$ ;
    }
    if ( $s$  is in  $H$ )
        return “yes”; // if  $D$  accepts  $s$ 
    else
        return “false”;
}

```

Figure:- Simulating DFA

Non Deterministic Automata (NFA)

FA is non deterministic, if there is more than one transition for each (state, input) pair. It is slower recognizer but it make take less spaces. An NFA is a five tuple (S, A, S_0, δ, F) where,

$S \rightarrow$ finite set of states

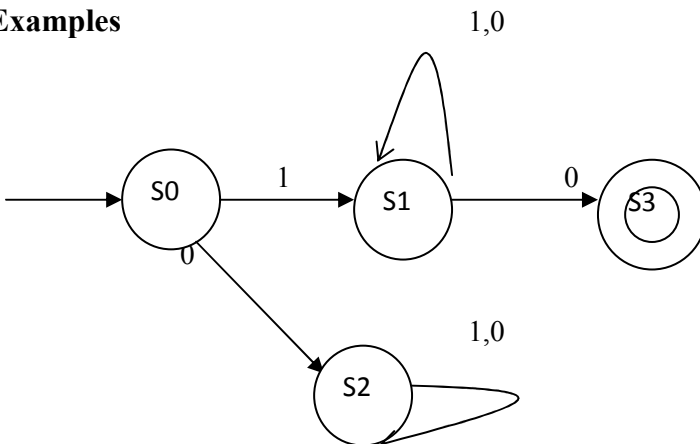
$A \rightarrow$ finite set of input alphabets

$S_0 \rightarrow$ starting state

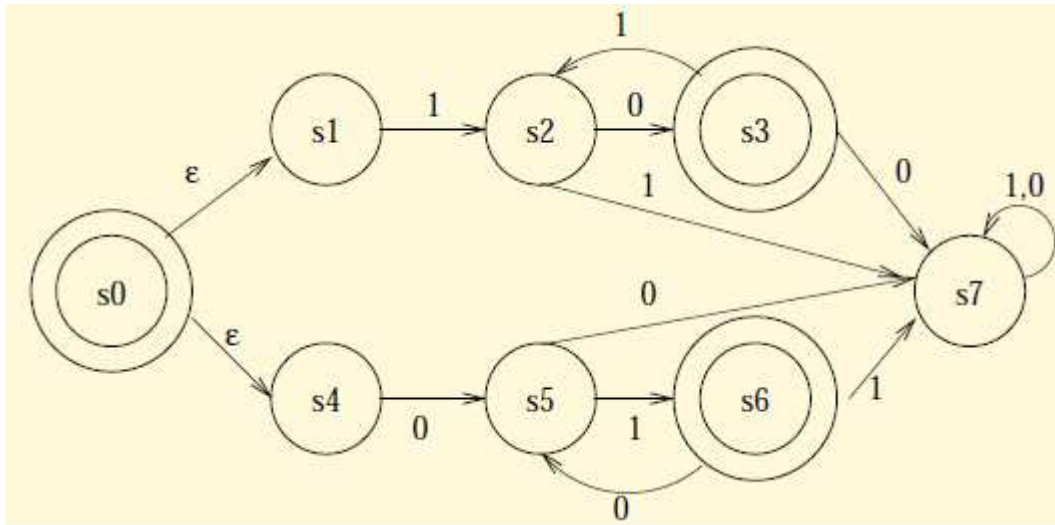
$\delta \rightarrow$ transition function i.e. $\delta: S \times A \rightarrow 2^{|S|}$

$F \rightarrow$ set of final states $F \subseteq S$

Examples



The above state machine is an NFA having a non – deterministic transition at state S1. On reading 0 it may either stay in S1 or goto S3 (accept). It's regular expression is $1(1 + 0)^*0$.



The above figure shows a state machine with ϵ moves that is equivalent to the regular expression $(10)^* + (01)^*$. The upper half of the machine recognizes $(10)^*$ and the lower half recognizes $(01)^*$. The machine non-deterministically moves to the upper half or the lower half when started from state s_0 .

Algorithm

$S = \epsilon\text{-closure}(\{S_0\})$ // set of all states that can be accessed from S_0 by ϵ -transitions

$c = \text{getchar}()$

while($c \neq \text{eof}$)

{

$S = \epsilon\text{-closure}(\text{move}(S, c))$ // set of all states that can be accessible from a state in S by a transition on c

$c = \text{getchar}()$

}

if ($S \cap F \neq \emptyset$) then

return "YES"

else

return "NO"

Figure :- Simulating NFA

Reducibility

1. NFA to DFA
2. RE to NFA
3. RE to DFA

NFA to DFA (sub set construction)

This sub set construction is an approach for an algorithm that constructs DFA from NFA, that recognizes the same language. Here there may be several accepting states in a given subset of nondeterministic states. The accepting state corresponding to the pattern listed first in the lexical analyser generator specification has priority. Here also state transitions are made until a state is reached which has no next state for the current input symbol. The last input position at which the DFA entered an accepting state gives the lexeme. We need the following operations.

- ϵ -closure(S) \rightarrow the set of NFA states reachable from NFA state S on ϵ -transition
- ϵ -closure(T) \rightarrow the set of NFA states reachable from some NFA states S in T on ϵ -transition
- $\text{Move}(T, a) \rightarrow$ the set of NFA states to which there is a transition on input symbol a from NFA state S in T .

Subset Construction Algorithm

Put ϵ -closure(S_0) as an unmarked state in $DStates$

While there is an unmarked state T in $DStates$ **do**

Mark T

For each input symbol $a \in A$ **do**

$U = \epsilon$ -closure(move(T, a))

If U is not in $DStates$ **then**

Add U as an unmarked state to $DStates$

End if

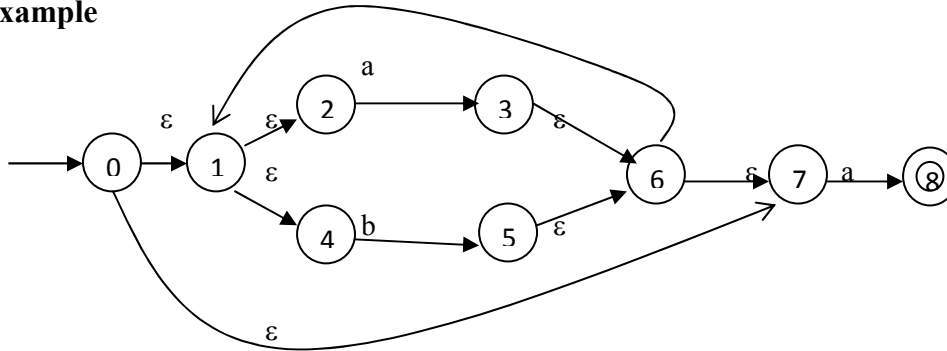
$DTran[T, a] = U$

End do

End do

- $DStates$ is the set of states of the new DFA consisting of sets of states of the NFA
- $DTran$ is the transition table of the new DFA
- A set of $DStates$ is an accepting state of DFA if it is a set of NFA states containing at least one accepting state of NFA
- The start state of DFA is ϵ -closure(S_0)

Example



$S_0 = \epsilon$ -closure($\{0\}$) = $\{0, 1, 2, 7\} \rightarrow S_0$ into $DStates$ as an unmarked state

Mark S_0

ϵ -closure(move(S_0, a)) = ϵ -closure($\{3, 8\}$) = $\{1, 2, 3, 4, 6, 7, 8\} = S_1 \rightarrow S_1$ into $DStates$ as an unmarked state

ϵ -closure(move(S_0, b)) = ϵ -closure($\{5\}$) = $\{1, 2, 4, 5, 6, 7\} = S_2 \rightarrow S_2$ into $DStates$ as an unmarked state

$DTran[S_0, a] \leftarrow S_1$

$DTran[S_0, b] \leftarrow S_2$

Mark S_1

ϵ -closure(move(S_1, a)) = ϵ -closure($\{3, 8\}$) = $\{1, 2, 3, 4, 6, 7, 8\} = S_1$

ϵ -closure(move(S_1, b)) = ϵ -closure($\{5\}$) = $\{1, 2, 4, 5, 6, 7\} = S_2$

$DTran[S_1, a] \leftarrow S_1$

$DTran[S_1, b] \leftarrow S_2$

Mark S_2

ϵ -closure(move(S_2, a)) = ϵ -closure($\{3, 8\}$) = $\{1, 2, 3, 4, 6, 7, 8\} = S_1$

ϵ -closure(move(S_2, b)) = ϵ -closure($\{5\}$) = $\{1, 2, 4, 5, 6, 7\} = S_2$

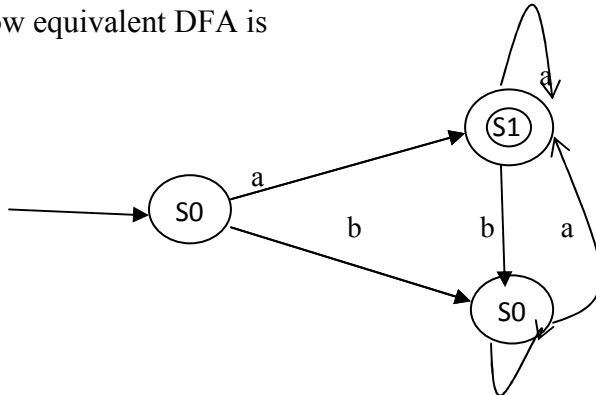
$DTran[S_2, a] \leftarrow S_1$

$DTran[S_2, b] \leftarrow S_2$

S_0 is the start of DFA, since 0 is a member of $S_0 = \{0, 1, 2, 4, 7\}$

S_1 is an accepting state of DFA, since 8 is a member of $S_1 = \{1, 2, 3, 4, 6, 7, 8\}$

Now equivalent DFA is



RE to NFA (Thomson's Construction)

Input \rightarrow RE, r , over alphabet A

Output \rightarrow ϵ -NFA accepting $L(r)$

Procedure \rightarrow process in bottom up manner by creating ϵ -NFA for each symbol in A including ϵ . Then recursively create for other operations as shown below

1. For a in A and ϵ , both are RE and constructed as

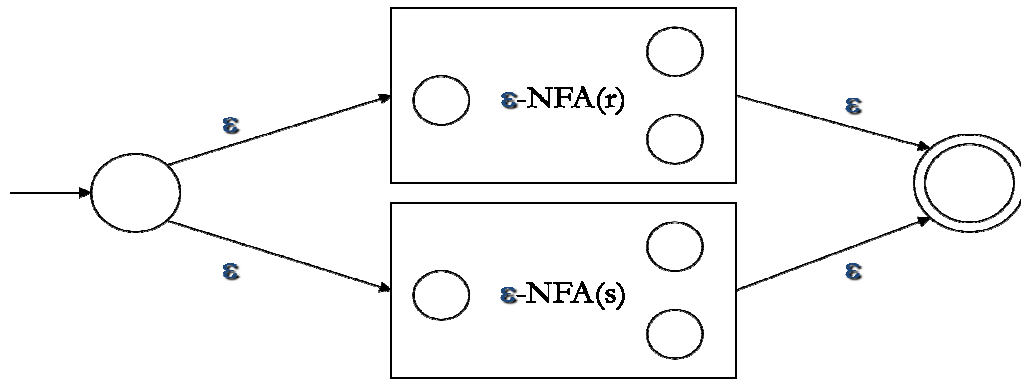


2. For REs r and s , $r.s$ is RE too and it is constructed as

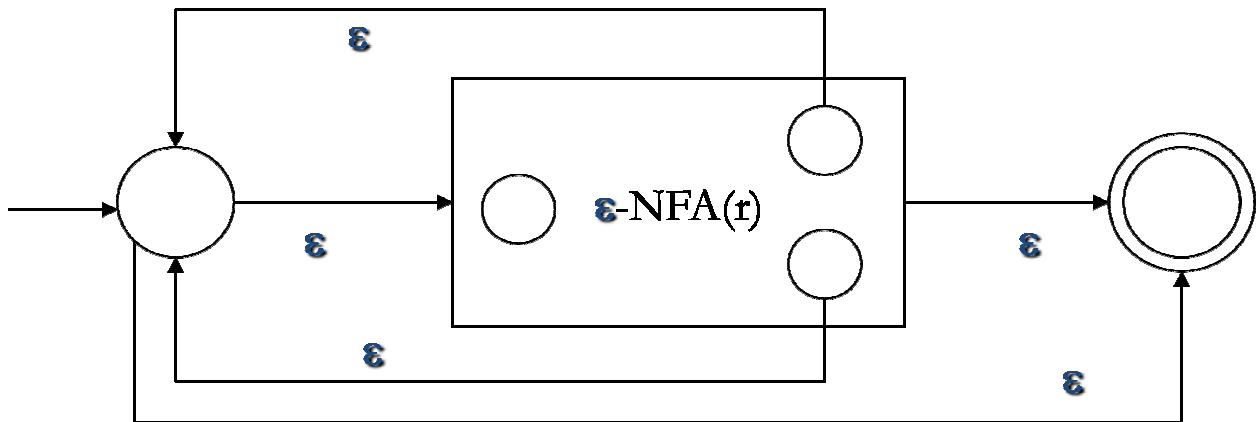


i.e. The start of r , becomes the start of $r.s$ and final state of s becomes the final state of $r.s$

3. For REs r and s , $r+s$ is RE too and it is constructed as



4. For REs r , r^* is RE too and it is constructed as

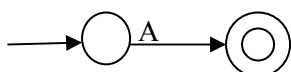


- For REs (r) , ϵ -NFA((r)) is ϵ -NFA(r)
- ϵ -NFA(r) has at most twice as many state as number of symbols and operators in r since each construction takes a tmost two new states.
- ϵ -NFA(r) has exactly one start and one accepting state.
- Each state of ϵ -NFA(r) has either one outgoing transition on a symbol in A or at most two outgoing ϵ -transitions

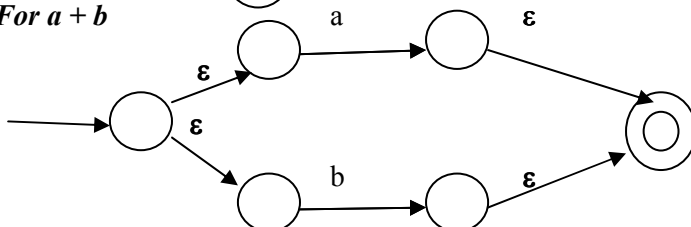
Example

$(a + b)^*a$ To NFA

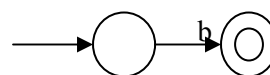
For a



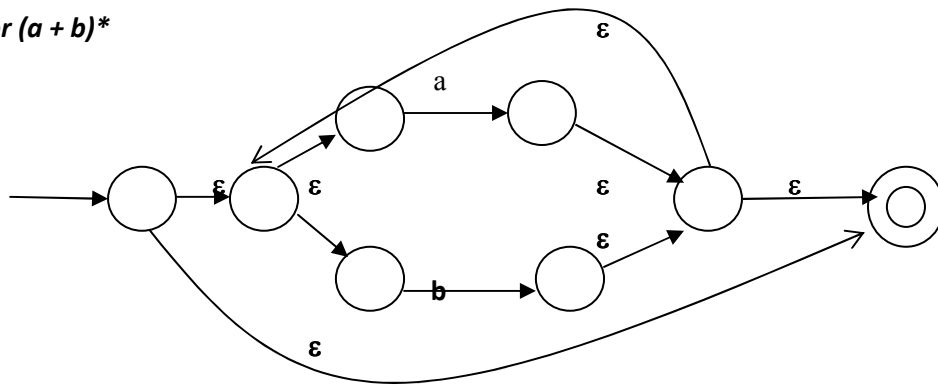
For $a + b$



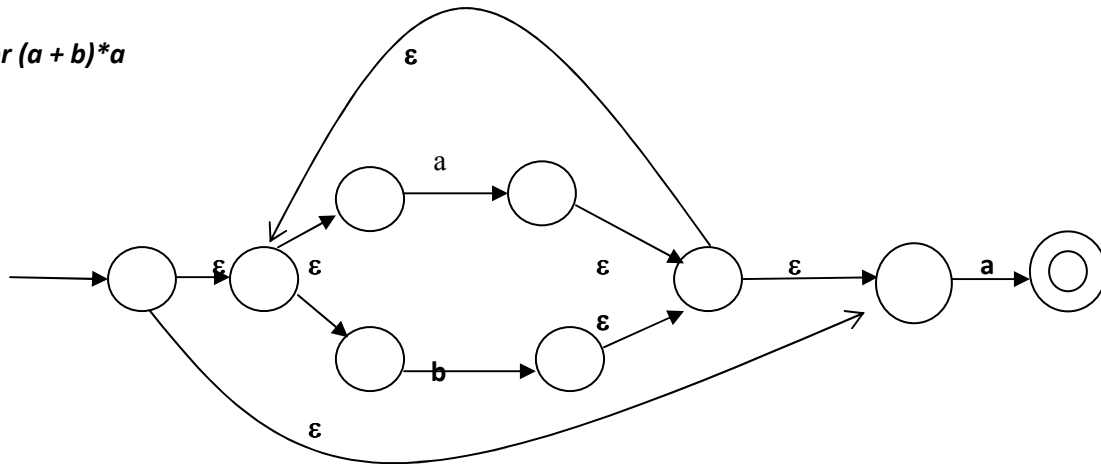
For b



For $(a + b)^*$



For $(a + b)^*a$



RE to DFA

Important States

The state s in ϵ -NFA is an important state if it has no null transition. In optimal state machine all states are important states

Augmented Regular Expression

ϵ -NFA created from RE has exactly one accepting state and accepting state is not important state since there is no transition so by adding special symbol $\#$ on the RE at the rightmost position, we can make the accepting state as an important state that has transition on $\#$. Now the accepting state is there in optimal state machine. The RE $(r)\#$ is called augmented RE.

Procedure

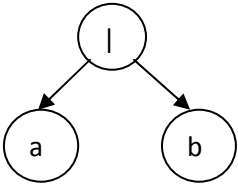
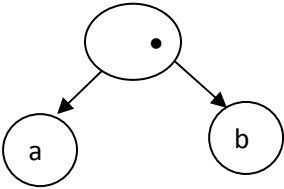
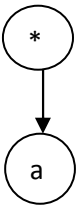
1. Convert the given expression to augmented expression by concatenating it with " $\#$ " i.e $(r) \rightarrow (r)\#$.
2. Construct the syntax tree of this augmented regular expression. In this tree, all the operators will be inner nodes and all the alphabet, symbols including " $\#$ " will be leaves.
3. Numbered each leaves.

4. Traverse tree to construct *nullable*, *firstpos*, *lastpos* and *followpos*.
5. Construct the DFA from so obtained *followpos*.

▪ **Some Definitions**

- **nullable(*n*)**: If the subtree rooted at *n* can have valid string as ϵ , then nullable(*n*) is true, false otherwise.
- **firstpos(*n*)**: The set of all the positions that can be the first symbol of the substring rooted at *n*.
- **lastpos(*n*)**: The set of all the positions that can be the last symbol of the substring rooted at *n*.
- **followpos(*i*)**: The set of all positions that can follow *i* for valid string of regular expression. We calculate followpos, we need above three functions.

Rule to evaluate *nullable*, *firstpos* and *lastpos*

Node- <i>n</i>	nullable(<i>n</i>)	firstpos(<i>n</i>)	lastpos(<i>n</i>)
leaf labeled ϵ	True	{}	{}
non null leaf position <i>i</i>	False	{ <i>i</i> }	{ <i>i</i> }
	nullable(<i>a</i>) or nullable(<i>b</i>)	$\text{firstpos}(a) \cup \text{firstpos}(b)$	$\text{lastpos}(a) \cup \text{lastpos}(b)$
	nullable(<i>a</i>) and nullable(<i>b</i>)	if nullable(<i>a</i>) is true then $\text{firstpos}(a) \cup \text{firstpos}(b)$ else firstpos(<i>a</i>)	if nullable(<i>b</i>) is true then $\text{lastpos}(a) \cup \text{lastpos}(b)$ else lastpos(<i>b</i>)
	True	firstpos(<i>a</i>)	lastpos(<i>a</i>)

Computation of *followpos*

Algorithm

```

for each node  $n$  in the tree do
    if  $n$  is a cat-node with left child  $c_1$  and right child  $c_2$  then
        for each  $i$  in  $\text{lastpos}(c_1)$  do
             $\text{followpos}(i) = \text{followpos}(i) \cup \text{firstpos}(c_2)$ 
        end do
    else if  $n$  is a star-node then
        for each  $i$  in  $\text{lastpos}(n)$  do
             $\text{followpos}(i) = \text{followpos}(i) \cup \text{firstpos}(n)$ 
        end do
    end if
end do

```

Algorithm to create DFA from RE

1. Create a syntax tree of $(r)\#$
2. Evaluate the functions nullable, firstpos, lastpos and followpos
3. Start state of DFA = $S = S_0 = \text{firstpos}(r)$, where r is the root of the tree, as an unmarked state
4. While there is unmarked state T in the state of DFA
 - Mark T
 - for each** input symbol a in A **do**
 - Let s_1, s_2, \dots, s_n are positions in S and symbols in those positions is a
 - $S' \leftarrow \text{followpos}(s_1) \cup \dots \cup \text{followpos}(s_n)$
 - $\text{Move}(S, a) \leftarrow S'$
 - if** (S' is not empty and not in the states of DFA)
 - Put S' into states of DFA and unmarked it
 - end if**
 - end do**

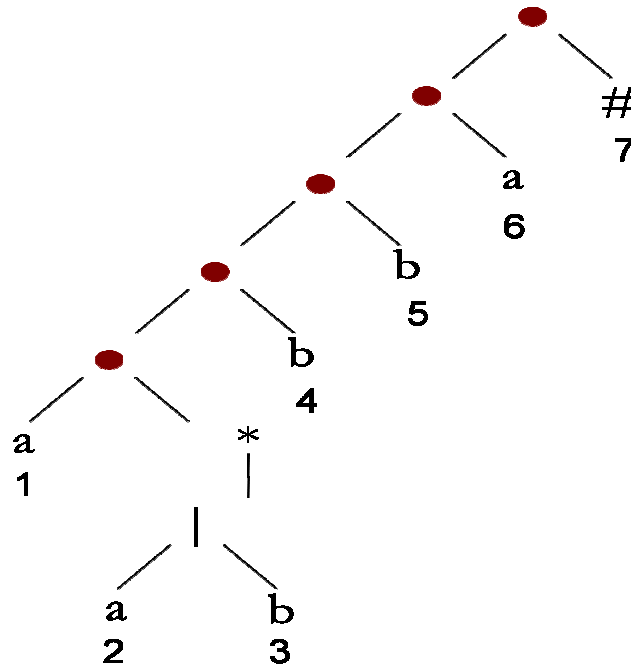
Note:

The start state of resulting DFA is $firstpos(root)$

The final states of DFA are all states containing the position of #.

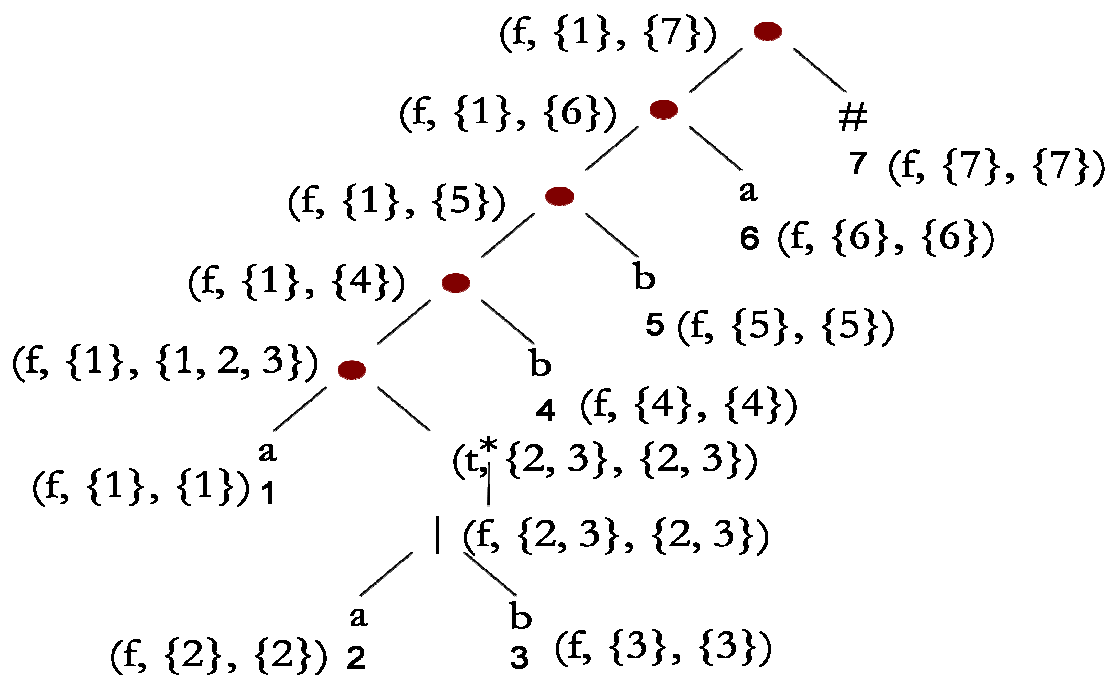
Example

Construct DFA from RE $a(a + b)^*bba\#$



Create a syntax tree

Calculate *nullable*, *firstpos* and *lastpos*



Calculate followpos

Using rules we get

followpos(1): {2, 3, 4}

followpos(2): {2, 3, 4}

followpos(3): {2, 3, 4}

followpos(4): {5}

followpos(5): {6}

followpos(6): {7}

followpos(7): -

Now,

Starting state = S1 = firstpos(root) = {1}

Mark S1

For a : followpos(1) = {2, 3, 4} = S2

For b : ϕ

Mark S2

For a : followpos(2) = {2, 3, 4} = S2 (because among {2, 3, 4}, position of a is 2)

For b : followpos(3) \cup followpos(4) = {2, 3, 4, 5} = S3 (because among {2, 3, 4}, position of b is

3 and 4)

Mark S3

For a : followpos(2) = {2, 3, 4} = S2

For b : followpos(3) \cup followpos(4) \cup followpos(5) = {2, 3, 4, 5, 6} = S4

Mark S4

For a : followpos(2) \cup followpos(6) = {2, 3, 4, 7} = S5 (since it contains 7 i.e. #, so it is accepting state)

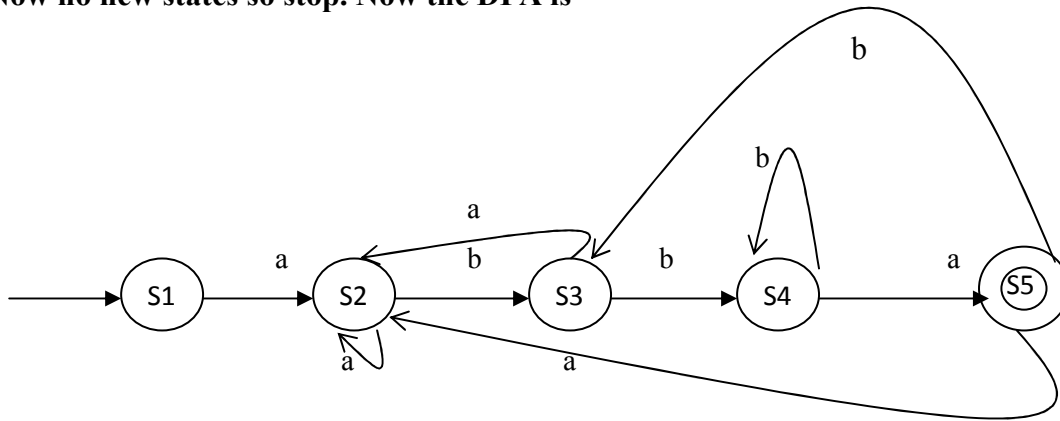
For b : followpos(3) \cup followpos(4) \cup followpos(5) = {2, 3, 4, 5, 6} = S4

Mark S5

For a : followpos(2) = {2, 3, 4} = S2

For b : followpos(3) \cup followpos(4) = {2, 3, 4, 5} = S3

Now no new states so stop. Now the DFA is

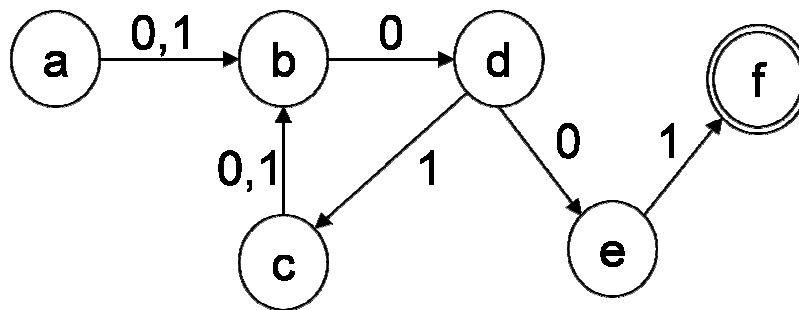


State Minimization in DFA

DFA minimization refers to the task of transforming a given deterministic finite automaton (DFA) into an equivalent DFA which has minimum number of states. Two states p and q are called **equivalent** if for all input string s , $\delta(p, w)$ is an accepting state iff $\delta(q, w)$ is an accepting state., otherwise **distinguishable** states. We say that, string w distinguishes state s from state t if, by starting with DFA M in state s and feeding it input w , we end up in an accepting state, but starting in state t and feeding it with same input w , we end up in a non accepting state, or vice – versa. It finds the states that can be distinguished by some input string. Each group of states that cannot be distinguished is then merged into a single state.

Procedure

1. So partition the set of states into two partition a) set of accepting states and b) set of nonaccepting states.
2. Split the partition on the basis of distinguishable states and put equivalent states in a group
3. To split we process the transition from the states in a group with all input symbols. If the transition on any input from the states in a group is on different group of states then they are distinguishable so remove those states from the current partition and create groups.
4. Process until all the partition contains equivalent states only or have single state.

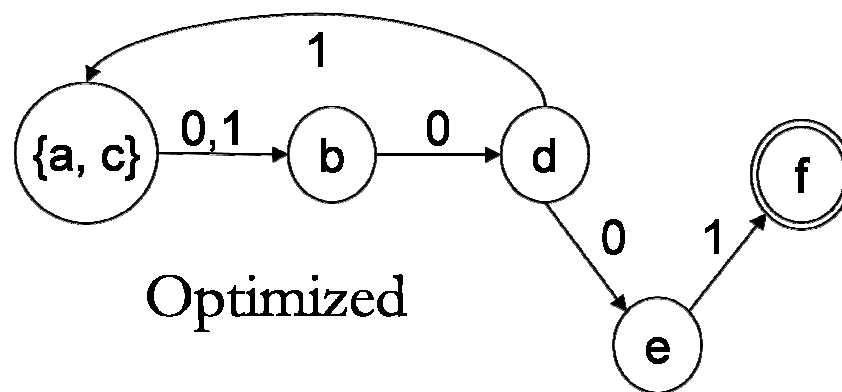
Example**Optimize the DFA**

Partition 1: $\{\{a, b, c, d, e\}, \{f\}\}$ with input 0; $a \rightarrow b$ and $b \rightarrow d$, $c \rightarrow b$, $d \rightarrow e$ all transition in same group. with input 1; $e \rightarrow f$ (different group) so e is distinguishable from others.

Partition 2: $\{\{a, b, c, d\}, \{e\}, \{f\}\}$ with input 0; $d \rightarrow e$ (different group).

Partition 3: $\{\{a, b, c\}, \{d\}, \{e\}, \{f\}\}$ with input 0; $b \rightarrow d$ (Watch!!!)

Partition 4: $\{\{a, c\}, \{b\}, \{d\}, \{e\}, \{f\}\}$ with both 0 and 1 $a, c \rightarrow b$ so no split is possible here, a and c are equivalent.

**Space Time Tradeoffs: NFA Vs DFA**

- Given the RE r and the input string s to determine whether s is in $L(r)$ we can either construct NFA and test or we can construct DFA and test for s after NFA is constructed from r .
- ϵ - NFA (for NFA only constant time differs)
 - Space complexity: $O(|r|)$ (at most twice the number of symbols and operators in r).

- Time complexity: $O(|r|^*|s|)$ (test s , there may be $O(|s|)$ test for each possible transition).
- DFA
 - Space complexity: $O(2^{|r|})$ (ϵ - NFA construction and then subset construction).
 - Time complexity: $O(|s|)$ (single transition so linear test for s).

If we can create DFA from RE by avoiding transition table, then we can improve the performance

Exercises

1. Given alphabet $A = \{0, 1\}$, write the regular expression for the following
 - a. Strings that can either have sub strings 001 or 100
 - b. String where no two 1s occurs consecutively
 - c. String which have an odd numbers of 0s
 - d. String which have an odd number of 0s and even number of 1s
 - e. String that have at most 2 0s
 - f. String that have at least 3 1s
 - g. String that have at most two 0s and at least three 1s
2. Write regular definition for specifying integer number, floating number, integer array declaration in C.
3. Convert the following regular expression first into NFA and DFA.
 - a. $0 + (1 + 0)^*00$
 - b. $\text{zero} \rightarrow 0$
 $\text{one} \rightarrow 1$
 $\text{bit} \rightarrow \text{zero} + \text{one}$
 $\text{bits} \rightarrow \text{bit}^*$
4. Write an algorithm for computing $\varepsilon\text{-closure}(s)$ of any state s in NFA.
5. Converse the following RE to DFA
 - a. $(a + b)^*a$
 - b. $(a + \varepsilon) b c^*$
6. Describe the languages denoted by the following RE
 - a. $0(0 + 1)^*0$
 - b. $((\varepsilon + 0)1^*)^*$
 - c. $(0 + 1)^*0(0 + 1)(0 + 1)$
 - d. $0^*10^*10^*10^*$
 - e. $(00 + 11)^*((01 + 10)(00 + 11)^*(01 + 10)(00 + 11)^*)^*$
7. Construct NFA from following RE and trace the algorithm for ababbab
 - a. $(a + b)^*$
 - b. $(a^* + b^*)^*$
 - c. $((\varepsilon + a)b^*)^*$

- d. $(a + b)^* abb(a + b)^*$
8. Show that following RE are equivalent by showing their minimum state DFA
- a. $(a + b)^*$
- b. $(a^* + b^*)^*$
- c. $((\epsilon + a)b^*)^*$

Bibliography

- Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools. 2008
- Notes from CDCSIT, TU of Samujjwal Bhandari and Bishnu Gautam
- Manuals of Srinath Srinivasa, Indian Institute of Information Technology, Bangalore

Chapter 4: Syntax Analysis(Parsing)

The Role of a Parser: The second phase of the compilation process is syntax analysis commonly known as parsing. A parser obtains the tokens from the lexical analyzer and analyzes syntactically according to the grammar of the source language whether the string can be generated or not from the grammar i.e. the parser works with the lexical analyzer as shown in figure below.

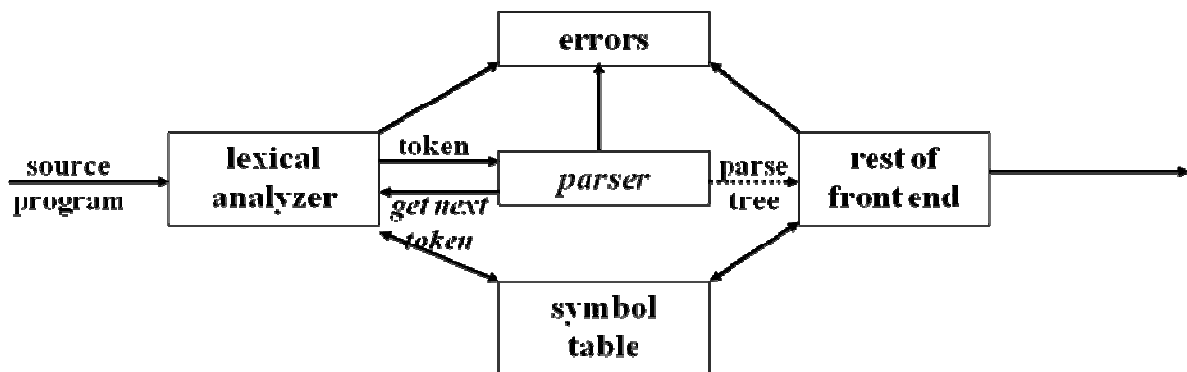


Figure: Position of parser in compiler model

A syntax analyzer(parser) is to analyze the source program based on the definition of its syntax. it works in lock-up step with the lexical analyzer(scanner) and responsible for creating a parse tree out of the source code.

- A parser implements a Context Free Grammar.
- Besides the checking of syntax the parser is responsible to report the syntax errors.
- A parser is also responsible to invoke semantic actions
 - for static semantics checking e.g. type checking of expressions, functions etc
 - for syntax directed translation of the source code to an intermediate representation
 - The possible intermediate representations outputs are
 - Abstract syntax tree
 - control-flow graphs(CFGs) with triples, three address code or register transfer list notations

Syntax Error Handling :

A good compiler should assist in identifying and locating errors. Programs may contain errors at different levels such as:

- Lexical errors: misspelling an identifier, keywords or operators. The compiler can easily recover and continue from those types of errors.
- Syntax errors: e.g. an expression with unbalanced parentheses or operator misplaced etc. Such errors are most important and can almost always be recovered.
- Semantic errors(static type): important and can sometimes be recovered.

- Semantic errors(Dynamic) : Hard or impossible to detect at compile time, runtime check is needed.
- Logical errors: hard and impossible to detect by compiler. e.g. infinite loops, recursive calls etc.

Context Free Grammar:

Programming languages usually have recursive structures that can be defined by context free grammars. A CFG is defined by 4-tuples as $G=(V,T,P,S)$ where V is set of variable symbols, T stands for set of terminal symbols, P stands for set of productions(rules) and S is a special variable called start symbol from which the derivation of each string is started.

e.g.

$E \rightarrow E A E \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

Here, E and A are non terminals with E as start symbol and other symbols are terminals.

Derivation: Process of obtaining the terminal strings from the start symbol of the grammar.

- The term $\alpha \Rightarrow \beta$ denotes that β can be derived from α by applying a production of α .
- The term $\alpha \Rightarrow^* \beta$ denotes that β can be derived by 0 or more production rules from α
- The term $\alpha \Rightarrow^+ \beta$ denotes that β can be derived by 1 or more production rules from α
- If β can be derived by replacing the left most(right-most) non terminal in every derivation steps, then it is called left-most(right-most) derivation of α

Leftmost: Replace the leftmost non-terminal symbol

$E \Rightarrow E A E \Rightarrow id A E \Rightarrow id * E \Rightarrow id * id$

Rightmost: Replace the leftmost non-terminal symbol

$E \Rightarrow E A E \Rightarrow E A id \Rightarrow E * id \Rightarrow id * id$

EXAMPLE: $id * id$ is a sentence of above grammar, then the derivation is

$E \Rightarrow E A E \Rightarrow E * E \Rightarrow id * E \Rightarrow id * id$

$E \Rightarrow id * id$

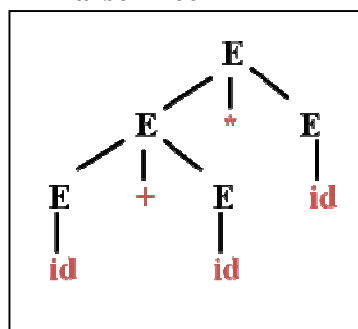
Parse Tree: A graphical representation of the derivation of any string from the grammar

Example:

Derivation:

$E \Rightarrow E * E$
 $\Rightarrow E + E * E$
 $\Rightarrow id + E * E$
 $\Rightarrow id + id * E$
 $\Rightarrow id + id * id$

Parse Tree



Ambiguity: If a same terminal string can be derived from the grammar using two or more distinct left-most derivation(or right-most) then the grammar is said to be ambiguous. i. e. from an ambiguous grammar, we can get two or more distinct parse tree for the same terminal string.

Left –recursion: A left recursive grammar is one that has rules like $A \Rightarrow A\alpha$, for some α . Top-Down parsing can't reconcile this type of grammar, since it could consistently make choice which wouldn't allow termination and the parser moves to an infinite loop like as $A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \dots$ etc. for grammar $A \rightarrow A\alpha \mid \beta$

The left recursion from the grammar can be removed as:

Left recursive grammar:

$$A \rightarrow A\alpha \mid \beta$$

To the following:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

In general,

The left recursive rules like

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ where no β_i begins with A. This can be converted without left recursion as

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

Example:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

Here E and T productions contains the left recursion so removing the recursion the grammar without the left recursion will be as

$$E \rightarrow TE'$$

$$E' \rightarrow + TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow * FT' \mid \epsilon$$

$$F \rightarrow (E) \mid id$$

Parsing

Given a stream of tokens, parsing involves the process of reducing them into a non-terminal. The input string is said to represent the non-terminal it was reduced to.

Parsing can be either top-down or bottom up.

- Top down parsing involves generating the string starting from the first non-terminal (start-symbol) and repeatedly applying the production of the grammar.
- Bottom-up parsing involves repeatedly re-writing the input string until it ends up in the start non-terminal.

Following are the Top-down parsing algorithms

1. Recursive Descent Parsing
2. Non-recursive predictive parsing

Consider the grammar

$$S \rightarrow cAd$$

$$A \rightarrow ab \mid a$$

and the input string is $w = cad$, The recursive descent parsing algorithm can be described as below.

Consider the parsing string $\alpha = S$ (start Symbol)

1. Let $iptr$ be the index of the input string and $optr$ be the index of the output string, and initially
 $iptr = optr = 0$

Input: $iptr(cad)$; output: $optr(S)$

2. while $\alpha[optr]$ is non-terminal, expand the non-terminal with its first production rule.

Input: $iptr(cad)$; Output: $(optr)cAd$

3. while $w[iptr] = \alpha[optr]$, increment both $iptr$ and $optr$. If end of string is reached, then success.

Input: $c(iptr)ad$; Output: $c(optr)Ad$

4. The while loop above stops if,

- a. a non terminal is encountered in α
- b. end of string is reached or
- c. if $w[iptr] \neq \alpha[optr]$

5. if (a) is true, then goto step 2 and expand the non-terminal with the first production.

Input: $c(iptr)ad$; Output: $c(optr)abd$

Input: $ca(iptr)d$; Output: $ca(optr)bd$

6. If (b) is true then exit with success. If (c) is true then revert $iptr$ and $optr$ to the place they were the last expansion, and replace the non-terminal with the next production rule.

Input: $c(iptr)ad$; Output: $c(optr)Ad$

Input: $c(iptr)ad$; Output: $c(optr)ad$

Input: $ca(iptr)d$; Output: $ca(optr)d$

Input: $cad(iptr)$ Output: $cad(optr)$

Exercise:

Given the following grammar

$R \rightarrow idS / (R) / S$

$S \rightarrow +RS / .RS / * S / \epsilon$

Then token *id* can be one of {a,b}. Determine whether the following strings are in the grammar using recursive descent parsing.

1. $a.(a+b)*.b$
2. $(b.a.b)^*$
3. $a.b..b.a$

Non-recursive Predictive parsing

A recursive descent parser always chooses the first available production whenever encountered by a non-terminal. This is inefficient and causes a lot of back-tracking. It also suffers from the left-recursion problem. A predictive parser tries to predict which production produces the least chances of a backtracking and infinite looping by choosing the proper production for the derivation of the input string.

Lookahead predictive parser:

- Predictive parsing relies on information about what first symbol can be generated from a production.
- If the first symbol of a production can be a non-terminal then the non-terminal has to be expanded till we get a set of terminals.
- For any given strings of terminals and non-terminals α , $FIRST(\alpha)$ defines the set of all terminals that can be generated from α .

Consider the rules:

$type \rightarrow simple / id / array [simple] \text{ of } type$

$simple \rightarrow integer / char$

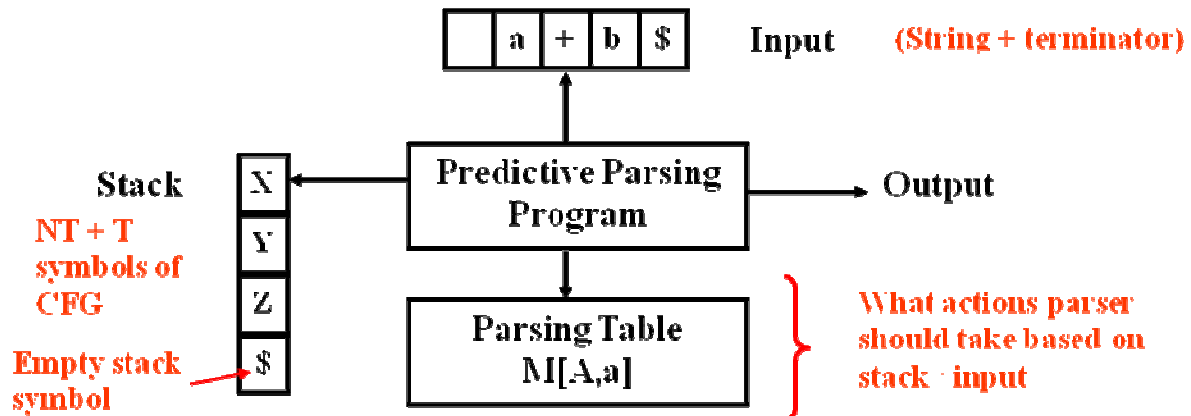
then, the FIRST of each symbol can be as:

$FIRST(type) = \{integer, char, id, array\}$

$FIRST(simple) = \{integer, char\}$

For the each terminal symbol 'a' the $FIRST(a) = \{a\}$

Non-Recursive predictive parsing



- Non-recursive predictive parsing is a table driven predictive parser comprises of an input buffer, stack, parsing table and an output stream.
- The input buffer and the stack are delimited by a special '\$' symbol that denotes the end of stack or buffer.
- The parsing table is made of entries of the form $M[X,a] = \alpha$, which says that if the stack points to non-terminal X and input buffer points to symbol 'a' then X has to be replaced by α , here α may be a set of terminals and non terminals or error.
- The program for the parser behaves as follows:
 1. If the stack top symbol and the input symbol is '\$', the parser halts and announces successful parsing.
 2. If the stack top symbol and the input symbol matches which is not '\$', the parser pops the stack and advances the input pointer to the next symbol.
 3. If the stack top symbol X is a non-terminal, then program consults entry $M[X,a]$ of parsing table M . This entry will be either X production or an Error.
 - If it is a X production $\{X \rightarrow UVW\}$, it replaces the top of the stack X by WVU (U becomes the new top symbol)
 - If $M[X,a] = \text{Error}$, the parser calls an error recovery routine.

So the algorithm for the parser can be explained as below.

Input: A string w and a parsing table M for the grammar G .

Method: Initially the parser is in configuration in which SS on the stack with S on the top and $w\$$ in input buffer.

1. Set ip to the first symbol of the input string.
2. Set the stack to SS where S is the start symbol of the grammar G .
3. Let X be the top stack symbol and 'a' be the symbol pointed by ip then

Repeat

- a. If X is a terminal or '\$' then
 - i. If $X = a$ then pop X from the stack and advance ip
 - ii. else error()
- b. Else
 - i. If $M[X,a] = Y_1Y_2Y_3.....Y_k$ then

1. Pop X from Stack
2. Push $Y_k, Y_{k-1}, \dots, Y_2, Y_1$ on the stack with Y_1 on top.
3. Output the production $X \rightarrow Y_1 Y_2 Y_3 \dots Y_k$

Until $X = \$$.

Example:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

Given the parsing table for the grammar G as:

Non Terminals	Inputs					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

For the input string $id+id*id$, the moves made by predictive parser will be as follows

STACK	INPUT	OUTPUT
\$E	$id + id * id \$$	
\$E'T	$id + id * id \$$	$E \rightarrow TE'$
\$E'T'F	$id + id * id \$$	$T \rightarrow FT'$
\$E'T'id	$id + id * id \$$	$F \rightarrow id$
\$E'T'	$+ id * id \$$	
\$E'	$+ id * id \$$	$T' \rightarrow \epsilon$
\$E'T+	$+ id * id \$$	$E' \rightarrow +TE'$
\$E'T	$id * id \$$	
\$E'T'F	$id * id \$$	$T \rightarrow FT'$
\$E'T'id	$id * id \$$	$F \rightarrow id$
\$E'T'	$* id \$$	
\$E'T'F*	$* id \$$	$T' \rightarrow *FT'$
\$E'T'F	$id \$$	
\$E'T'id	$id \$$	$F \rightarrow id$
\$E'T'	$\$$	
\$E'	$\$$	$T' \rightarrow \epsilon$
\$	$\$$	$E' \rightarrow \epsilon$

Expend Input

The leftmost derivation for the example is as follows:

$$\begin{aligned}
 E &\Rightarrow TE' \Rightarrow FT'E' \Rightarrow id\ T'E' \Rightarrow id\ E' \Rightarrow id + TE' \Rightarrow id + FT'E' \\
 &\Rightarrow id + id\ T'E' \Rightarrow id + id * FT'E' \Rightarrow id + id * id\ T'E' \\
 &\Rightarrow id + id * id\ E' \Rightarrow id + id * id
 \end{aligned}$$

Parsing Table:

The parsing table M comprises the entries of the form $M[X,a] = UVW$ meaning that if the top of the stack holds X and the input symbol 'a' is read, then X should be replaced by UVW.

- The construction of the parsing table is aided by two functions : FIRST and FOLLOW.
- If α is any string of grammar symbols, then $FIRST(\alpha)$ is the set of all terminals that begin the strings that can be derived from α . If $\alpha \Rightarrow \epsilon$, then ϵ is $FIRST(\alpha)$.
- For any non terminal A, FOLLOW(A) is the set of terminals 'a' that can appear immediately to the right of A in some sentential form. That is, there exist some rule of the form $S \Rightarrow \alpha A a \beta$, for some α and β .
- If A is right most symbol in some rule, then \$ is also in FOLLOW(A)

Computation of FIRST

1. For all terminals 'a', $First(a) = \{a\}$
2. For any non terminal X, if $X \rightarrow \epsilon$ is a production rule, add ϵ to $First(X)$ i.e $FIRST(X) = FIRST(X) \cup \{\epsilon\}$
3. If X is a non-terminal, for every rule of the form $X \rightarrow Y_1 Y_2 \dots Y_k$, update $First(X)$ by the following rules:
 - a. $FIRST(X) = FIRST(X) \cup FIRST(Y_1)$.
 - b. For all $1 < i < k$, $FIRST(X) = FIRST(X) \cup FIRST(Y_i)$ if ϵ is in $FIRST(Y_j)$ where $1 \leq j < i$. If ϵ is in $FIRST(Y_1) \cap FIRST(Y_2) \cap \dots \cap FIRST(Y_k)$ then $FIRST(X) = FIRST(X) \cup \{\epsilon\}$

Computation of FOLLOW

Apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the right end marker .
2. If there is a production of the form $A \rightarrow \alpha B \beta$, then every thing in $FIRST(\beta)$ except ϵ is placed in FOLLOW(B).
3. If $FIRST(\beta)$ in above case contains ϵ or if there is a rule of the form $A \rightarrow \alpha B$, then everything in FOLLOW(A) is in FOLLOW(B).

Example:

Consider the following grammar

$$R \rightarrow idS \mid (R)S$$

$$S \rightarrow +RS \mid .RS \mid *S \mid \epsilon$$

Here,

$$\text{FIRST}(R) = \{ \text{id}, (\}$$

$$\text{FIRST}(S) = \{ +, ., *, \epsilon \}$$

$$\text{FOLLOW}(R) = \{), +, ., *, \$ \}$$

$$\text{FOLLOW}(S) = \{), +, ., *, \$ \} \quad \text{from } R \rightarrow \text{id}S \text{ using the third rule above}$$

Another example:

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

$$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{ (, \text{id} \}$$

$$\text{FIRST}(E') = \{ +, \epsilon \}$$

$$\text{FIRST}(T') = \{ *, \epsilon \}$$

$$\text{FOLLOW}(E) = \{), \$ \} \quad \text{since } E \text{ is start symbol } \$ \text{ is follow of } E \text{ and }) \text{ comes in } \text{FOLLOW}(E) \text{ from the production } F \rightarrow (E)$$

$$\text{FOLLOW}(E') = \text{FOLLOW}(E) = \{), \$ \} \quad \text{from } E \rightarrow TE'$$

$$\text{FOLLOW}(T) = \{), +, \$ \} \quad \text{where }), \$ \text{ comes from } E \rightarrow TE' \text{ since } \epsilon \text{ is in } \text{FIRST}(E') \\ + \text{ comes from } \text{FIRST}(E')$$

$$\text{FOLLOW}(T') = \text{FOLLOW}(T) = \{), +, \$ \} \quad \text{since } T \rightarrow FT'$$

$$\text{FOLLOW}(F) = \{ *, +,), \$ \} \quad \text{where } * \text{ comes from } \text{FIRST}(T') \text{ in } T' \rightarrow FT', \text{ others come from } \text{FOLLOW}(T) \text{ in } T \rightarrow FT'$$

Example

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

The FIRST and FOLLOW:

FIRST:

$$\text{First}(S) = \{ i, a \}$$

$$\text{First}(S') = \{ e, \epsilon \}$$

$$\text{First}(E) = \{ b \}$$

Follow(S) – Contains \$, since S is start symbol

- Since $S \rightarrow iEtSS'$, put in $\text{First}(S')$ but not ϵ
- Since $S' \Rightarrow \epsilon$, Put in $\text{Follow}(S)$
- Since $S' \rightarrow eS$, put in $\text{Follow}(S')$ So....
- $\text{Follow}(S) = \{ e, \$ \}$
- $\text{Follow}(S') = \text{Follow}(S)$
- $\text{Follow}(E) = \{ t \}$

Construction of PARSING TABLE : Algorithm

For each production $A \rightarrow \alpha$ of the grammar do ,

1. For each terminal 'a' in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A,a]$.
2. (a) If ϵ is in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A,b]$ for every terminal 'b' in $\text{FOLLOW}(A)$.
(b) If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$ then add $A \rightarrow \alpha$ to $M[A,\$]$.
3. Make every undefined entry of M to be ERROR.

Example of parsing table for the grammar:

$R \rightarrow \text{id}S \mid (R)S$

$S \rightarrow +RS \mid .RS \mid *S \mid \epsilon$

The FIRST and FOLLOW as computed above are:

$\text{FIRST}(R) = \{ \text{id}, (\}$

$\text{FIRST}(S) = \{ +, ., *, \epsilon \}$

$\text{FOLLOW}(R) = \{), +, ., *, \$ \}$

$\text{FOLLOW}(S) = \{), +, ., *, \$ \}$

The parsing table is as below:

Non Terminal	Inputs						
	id	()	+	.	*	\$
R	$R \rightarrow \text{id}S$	$R \rightarrow (R)S$					
S			$S \rightarrow \epsilon$	$S \rightarrow +RS$ $S \rightarrow \epsilon$	$S \rightarrow .RS$ $S \rightarrow \epsilon$	$S \rightarrow *S$ $S \rightarrow \epsilon$	$S \rightarrow \epsilon$

In the above example,

- $\text{FIRST}(R)$ contains id and (so at the column with these inputs, corresponding productions of R has been entered.
- Also the $\text{FIRST}(S)$ contains $\{ +, ., *, \epsilon \}$ So for the symbols +, . and * the corresponding production with the first being that symbol are entered in table.
- The $\text{FIRST}(S)$ contains ϵ so for every symbols in $\text{FOLLOW}(S)$ the corresponding productions of S are entered .

In the above parsing table, some entries for $M[X,a]$ is multiply-defined. If the grammar is ambiguous than the parsing table may contain the multiple entries for some $M[X,a]$. So construction the parsing table for the grammar, we can decide whether the grammar is ambiguous or not.

Constructing Parsing Table another Example: Grammar, FIRST and FOLLOW

$S \rightarrow i E t S S' \mid a$	$\text{First}(S) = \{ i, a \}$	$\text{Follow}(S) = \{ e, \$ \}$
$S' \rightarrow e S \mid \epsilon$	$\text{First}(S') = \{ e, \epsilon \}$	$\text{Follow}(S') = \{ e, \$ \}$
$E \rightarrow b$	$\text{First}(E) = \{ b \}$	$\text{Follow}(E) = \{ t \}$

Construction of Parsing Table:

$S \rightarrow i E t S S'$	$S \rightarrow a$	$E \rightarrow b$
<u>$\text{First}(i E t S S') = \{ i \}$</u>	<u>$\text{First}(a) = \{ a \}$</u>	<u>$\text{First}(b) = \{ b \}$</u>
$S' \rightarrow e S$	$S \rightarrow \epsilon$	
<u>$\text{First}(e S) = \{ e \}$</u>	<u>$\text{First}(\epsilon) = \{ \epsilon \}$</u>	<u>$\text{Follow}(S') = \{ e, \\$ \}$</u>

Non-terminal	INPUT SYMBOL					
	a	b	e	i	t	\$
S	<u>$S \rightarrow a$</u>			<u>$S \rightarrow i E t S S'$</u>		
S'			<u>$S' \rightarrow \epsilon$</u> <u>$S' \rightarrow e S$</u>			<u>$S' \rightarrow \epsilon$</u>
E		<u>$E \rightarrow b$</u>				

Constructing Parsing Table – Another Example 3

<u>Grammar:</u>	<u>FIRST</u>	<u>FOLLOW</u>
$E \rightarrow T E'$		$\text{Follow}(E) = \{), \$ \}$
$E' \rightarrow + T E' \mid \epsilon$	$\text{First}(E, F, T) = \{ (, id \}$	$\text{Follow}(E') = \{), \$ \}$
$T \rightarrow F T'$	$\text{First}(E') = \{ +, \epsilon \}$	$\text{Follow}(F) = \{ *, +,), \$ \}$
$T' \rightarrow * F T' \mid \epsilon$	$\text{First}(T') = \{ *, \epsilon \}$	$\text{Follow}(T) = \{ +,), \$ \}$
$F \rightarrow (E) \mid id$		$\text{Follow}(T') = \{ +,), \$ \}$

Expression Example: $E \rightarrow T E' : \text{First}(T E') = \text{First}(T) = \{ (, id \}$ $M[E, (] : E \rightarrow T E'$ $M[E, id] : E \rightarrow T E' \quad \text{By Rule 1}$ $E' \rightarrow + T E' : \text{First}(+ T E') = +$ so $M[E', +] : E' \rightarrow + T E' \quad \text{By Rule 1}$ $T' \rightarrow * F T' : \text{First}(* F T') = *$

so, $M[T', *] : T' \rightarrow * FT'$ By Rule 1

$T \rightarrow FT' : \text{First}(FT') = \text{First}(F) = \{ (, id \}$

so $M[T, (] : T \rightarrow FT'$

$M[T, id] : T \rightarrow FT'$ By rule 1

$F \rightarrow (E) \mid id : \text{First}(F) = \{ (, id \}$ so by rule 1,

$M[F, (] : F \rightarrow (E)$

$M[F, id] : F \rightarrow id$

(by rule 2) $E' \rightarrow \epsilon : \epsilon \in \text{First}(\epsilon)$ $T' \rightarrow \epsilon : \epsilon \in \text{First}(\epsilon)$

$M[E',)] : E' \rightarrow \epsilon$ (2.a) $M[T', +] : T' \rightarrow \epsilon$ (2.a)

$M[E', \$] : E' \rightarrow \epsilon$ (2.a) $M[T',)] : T' \rightarrow \epsilon$ (2.b)

$M[T', \$] : T' \rightarrow \epsilon$ (2.b)

Non Terminals	Inputs					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Exercise: Construct the Parsing table for the grammar below using FIRST and FOLLOW.

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

LL(1) Grammar:

A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar. The first L in LL(1) corresponds to reading the input left to right and second 'L' corresponds to the left-most derivation. The 1 in the parenthesis corresponds to a maximum lookahead of 1 symbol.

- ✓ No ambiguous or left-recursive grammar is LL(1).
- ✓ There are no general rules to convert a non LL(1) grammar into a LL(1) grammar.
- ✓ The properties of LL(1) grammar are as below:

In any LL(1) grammar, if there exists a rule of the form $A \rightarrow \alpha \mid \beta$ where α and β , are distinct then,

1. For any terminal 'a', if a is in $\text{FIRST}(\alpha)$ then a is not in $\text{FIRST}(\beta)$
2. Either $\alpha \Rightarrow \epsilon$ or $\beta \Rightarrow \epsilon$, but not both.
3. If $\beta \Rightarrow \epsilon$, then α does not derive any string beginning with the terminal in $\text{FOLLOW}(A)$.

Exercise:

1. Show that the following grammar is ambiguous by constructing parsing table

$$S \rightarrow aSbS \mid bSaS \mid \epsilon$$

Bottom Up Parsing:

- ✓ Bottom up parsing attempts to construct a parse tree for an input string beginning at the leaves(the bottom) and working up towards root(top).
- ✓ The process of replacing a substring by a non-terminal in bottom up parsing is called reduction.
- ✓ The left-most reduction method corresponds to the right-most derivation of top-down parsing and the right-most reduction corresponds to the left-most derivation of top-down parsing.

Consider an example grammar

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

Now for string **abbcd**e can be reduced to S by following steps

abbcd

aAbcd (replace b by A using $A \rightarrow b$)

aAde (replace Abc by A using $A \rightarrow Abc$)

aABe (replace d by B using $B \rightarrow d$)

S (replace aABe by S using $S \rightarrow aABe$)

A substring that can be replaced by a non-terminal when it matches its right sentential form is called a **Handle**. So principle task of the bottom up parsing is to identify the handle that can be replaced by a non-terminal. Identifying the handle refers to the implementing the DFA that recognize the handle string.

Shift-Reduce Parsing

A simple bottom up parsing technique using a stack based implementation is shift-reduce parsing. A convenient way to implement a shift-reduce parser uses a stack to hold the grammar symbols and an input buffer to hold the input string **w**.

The method is described as:

1. Initially stack contains only the sentinel \$, and the input buffer contains the input string **w\$**.
2. While stack not equal to \$\$ do
 - a. While there is no handle at the top of stack, shift the input buffer and push the symbol onto stack.
 - b. If there is a handle on top of stack, then pop the handle and reduce the handle with its non terminal and push it on to stack.

Example:

For the above example string **abbcd**e\$, the shift reduce actions may be,

Stack	Input	Action
\$	abbcd	\$

\$a	bbcde\$	shift a
\$ab	bcde\$	shift b
\$aA	bcde\$	reduce $A \rightarrow b$
\$aAb	cde\$	shift b
\$aAbc	de\$	shift c
\$aA	de\$	reduce by $A \rightarrow Abc$
\$aAd	e\$	shift d
\$aAB	e\$	reduce by $B \rightarrow d$
\$aABe	\$	shift e
\$S	\$	reduce by $S \rightarrow aABe$

Conflict during Shift-Reduce Parsing

All grammars can not use the shift-reduce parsing since there may be conflicts during the parsing actions. For some grammars parser can not decide whether to shift or to reduce or can not decide which of several reduction to make. so the parsing action results in conflicts.

There are two kinds of conflicts in this method.

1. **Shift-reduce conflict:** The parser is not able to decide whether to shift or to reduce e.g. if $A \rightarrow ab \mid abcd$ are the productions of A and the stack contains \$ab and the input buffer contains cd\$, the parser cannot decide whether to reduce \$ab by non-terminal A or to shift two more symbols before reducing.
2. **Reduce-Reduce Conflict:** In this case the parser cannot decide which sentential form to use for reduction e. g. if $A \rightarrow bc$ and $B \rightarrow abc$ are the productions of grammar and the stack contains \$abc, the parser cannot decide whether to reduce it to \$aA or to reduce \$B.

The grammar that lead to the shift reduce parser into conflict are known as non-LR grammars. Shift reduce parser can be built successfully for LR grammars and operator grammars. The operator grammar has the property that no production right side is \in or has two adjacent terminals.

e.g. $E \rightarrow EAE \mid (E) \mid -E \mid id$ is not operator grammar.

$E \rightarrow E+E \mid E-E \mid E * E \mid (E) \mid -E \mid id$ is an operator grammar

LR Grammars

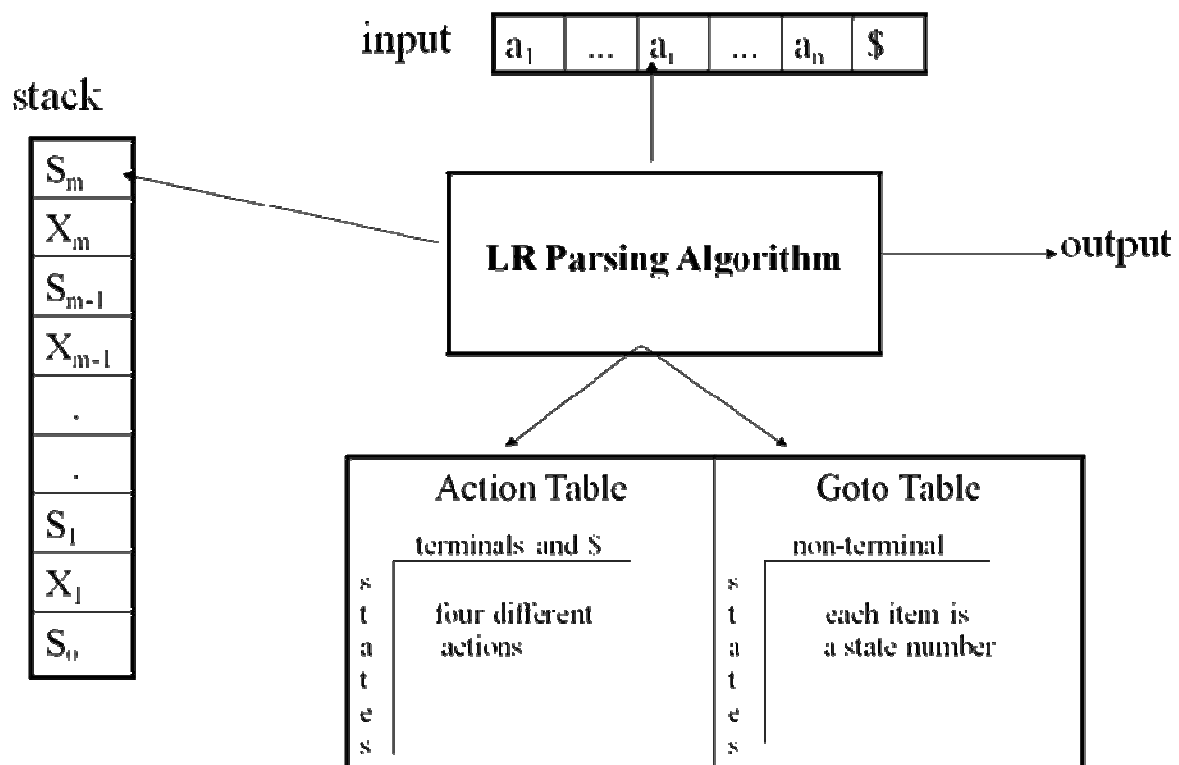
- ✓ The class of grammars called LR(k) grammars have the most efficient bottom up parser and can be implemented for almost any programming language.
- ✓ The first L stands for left to right scan of input buffer, the second R for a right most derivation (left-most reduction) and the k stands for the maximum number of input symbols of lookahead used for making parsing decision.
- ✓ If k is omitted, k is assumed to be 1.
- ✓ The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parser.

- ✓ It is difficult to write or trace LR parser by hand. Usually generators like Yacc(bison) is required to write LR parser.
- ✓ The LR parsing method is most general non-backtracking shift-reduce parsing method known.
- ✓ $LL(1) \text{ grammar} \subset LR(1) \text{ grammar}$.

LR-Parsers

- covers wide range of grammars.
- SLR – simple LR parser
- LR – most general LR parser
- LALR – intermediate LR parser (look-head LR parser)
- SLR, LR and LALR work same (they used the same algorithm), only their parsing tables are different.

Structure of LR Parser:



An LR parser comprises of a stack, an input buffer and a parsing table that has two parts: action and goto. Its stack comprises of entries of the form $s_0 X_1 s_1 X_2 s_2 \dots X_m s_m$ where every s_i is called a state and every X_i is a grammar symbol (terminal or non-terminal).

If top of the stack is S_m and input symbol is a , the LR parser consults $\text{action}[s_m, a]$ which can be one of four actions.

1. Shift s , where s is a state.

2. Reduce using production $A \rightarrow \beta$
3. Accept
4. Error

The function goto takes a state and grammar symbol as arguments and produces a state. The goto function forms the transition table for a DFA that recognizes the viable prefixes of the grammar.

Note: Viable prefixes are the set of prefixes of right sentential forms that can appear on the stack of the parser. i. e. it is a prefix of right sentential form that doesnot continue past the end of right most handle of that sentential form.

The configuration of the LR parser is a tuple comprising of the stack contents and the contents of the unconsumed input buffer. It is of the form

$(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m, a_i a_{i+1} \dots a_n \$)$

1. If $\text{action}[s_m, a_i] = \text{shift } s$, the parser executes a shift move entering the configuration $(s_0 X_1 s_1 X_2 s_2 \dots X_m s_m a_i s, a_{i+1} \dots a_n \$)$ shifting both a_i and the state s onto stack.
2. If $\text{action}[s_m, a_i] = \text{reduce } A \rightarrow \beta$, the parser executes a reduce move entering the configuration $(s_0 X_1 s_1 X_2 s_2 \dots X_{m-r} s_{m-r} A s, a_i a_{i+1} \dots a_n \$)$. Here $s = \text{goto}[s_{m-r}, A]$ and r is the length of handle β .

Note: if r is the length of β then the parser pops $2r$ symbols from the stack and push the state as on $\text{goto}[s_{m-r}, A]$. After reduction, the parser outputs the production used on reduce operation. $A \rightarrow \beta$

3. If $\text{action}[s_m, a_i] = \text{Accept}$, then parser accept i.e. parsing successfully complete and stop.
4. If $\text{action}[s_m, a_i] = \text{error}$ then call the error recovery routine.

Construction of the Parsing Table for the SLR parser (Simple LR)

- SLR parser are the simplest class of LR parser. Constructing a parsing table for action and goto involves building a state machine that can identify the handle.
- For building a state machine, we need to define the three terms: item, closure and goto

Item: An “item” is a production rule that contains a dot(.) somewhere in the right side of the production. For example, $A \rightarrow \cdot \alpha A \beta$, $A \rightarrow \alpha A \cdot \beta$, $A \rightarrow \alpha A \cdot \beta$, $A \rightarrow \alpha A \beta \cdot$ are items if there is a production $A \rightarrow \alpha A \beta$ in the grammar.

An item encapsulates what we have read until now and what we expect to read further from the input buffer.

The Closure operation:

If I is a set of items for a grammar G , then the $\text{closure}(I)$ is the set of items constructed from I using the following rules:

1. Initially, every item in I is added to closure(I).
 2. If $A \rightarrow \alpha B \beta$ is in closure(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \gamma$ to I if it is not already there.
- Apply this rule until no more new items can be added to closure(I).

Example:

$E' \rightarrow E$

$E \rightarrow E+T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$

If $I = \{E' \rightarrow \cdot E\}$ then closure(I) contains the following items:

{
 $E' \rightarrow \cdot E$,
 $E \rightarrow \cdot E+T$,
 $E \rightarrow \cdot T$,
 $T \rightarrow \cdot T * F$,
 $T \rightarrow \cdot F$,
 $F \rightarrow \cdot (E)$,
 $F \rightarrow \cdot id$
 }

Note:

- $E' \rightarrow \cdot E$ and all items whose dots are not at the left end(beginning of RSH) are called **kernel items**.
- All items with dot at the left end are non-kernel items except $E' \rightarrow \cdot E$ which is always known as **kernel items**.

The goto operation:

In any item I, for all production of the form $A \rightarrow \alpha X \beta$ that are in I,

Goto[I,X] is defined as the closure of all productions of the form $A \rightarrow \alpha B \beta$

In the example above, if $I_0 = \text{closure}(\{E' \rightarrow \cdot E\})$ then

$\text{goto}[I_0, E] = I_1 = \text{closure}(\{E' \rightarrow E \cdot, E \rightarrow \cdot E+T\})$ since the closure($\{E' \rightarrow \cdot E\}$) is $\{E' \rightarrow \cdot E, E \rightarrow \cdot E+T, E \rightarrow \cdot T, T \rightarrow \cdot T * F, T \rightarrow \cdot F, F \rightarrow \cdot (E), F \rightarrow \cdot id\}$

Similarly, $\text{goto}[I_0, T] = \text{closure}(\{E \rightarrow T \cdot, T \rightarrow T \cdot * F\})$

$\text{goto}[I_0, F] = \text{closure}(\{T \rightarrow F \cdot\})$ and so on.

The complete goto operation defines the DFA that identifies the handle.

Computing the Canonical LR(0) Collection

To create the SLR parsing tables for a grammar G, we will create the canonical LR(0) collection of the grammar G'.

Algorithm:

$C = \{ \text{closure}(\{S' \rightarrow \cdot S\}) \}$

repeat the followings until no more set of LR(0) items can be added to C.

for each I in C and each grammar symbol X

if goto(I,X) is not empty and not in C

 add goto(I,X) to C

The goto function is a DFA on the sets in C

Example: Compute the Canonical LR(0) items collection for the following grammar.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

Solution: The augmented grammar is ,

$E' \rightarrow E$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

$I_0 = \text{closure}(\{E' \rightarrow \cdot E\}) = \{$

$E' \rightarrow \cdot E,$
 $E \rightarrow \cdot E+T,$
 $E \rightarrow \cdot T,$
 $T \rightarrow \cdot T*F,$
 $T \rightarrow \cdot F,$
 $F \rightarrow \cdot (E),$
 $F \rightarrow \cdot id$
 $\}$

$\text{goto}[I_0, E] = \text{closure}(\{E' \rightarrow E \cdot, E \rightarrow E \cdot +T\}) = \{E' \rightarrow E \cdot, E \rightarrow E \cdot +T\} = I_1$

$\text{goto}[I_0, T] = \text{closure}(\{E \rightarrow T \cdot, T \rightarrow T \cdot *F\}) = \{E \rightarrow T \cdot, T \rightarrow T \cdot *F\} = I_2$

$\text{goto}[I_0, F] = \text{closure}(\{T \rightarrow F \cdot\}) = \{T \rightarrow F \cdot\} = I_3$

$\text{goto}[I_0, (] = \text{closure}(\{F \rightarrow (\cdot E\})$

$= \{F \rightarrow (\cdot E), E \rightarrow E \cdot +T, E \rightarrow T \cdot, T \rightarrow T \cdot *F, T \rightarrow F \cdot, F \rightarrow (\cdot E), F \rightarrow id\} = I_4$

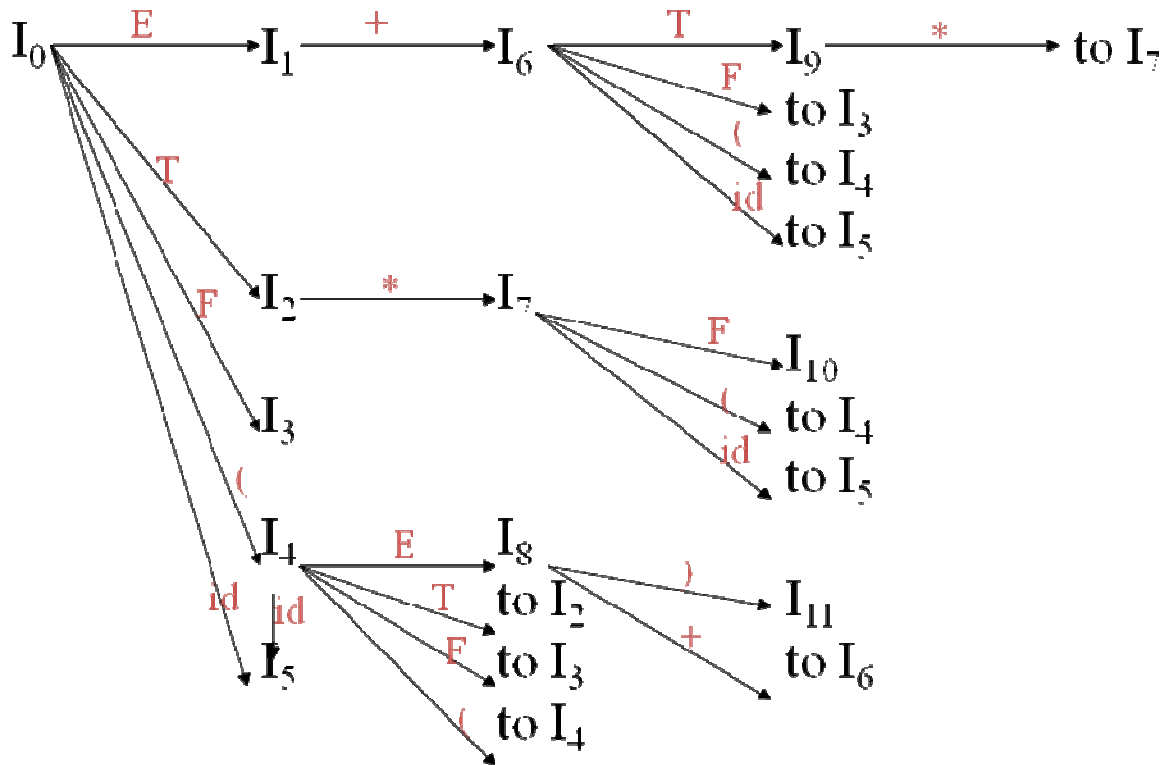
$\text{goto}[I_0, id] = \text{closure}(\{F \rightarrow id \cdot\}) = I_5$

$\text{goto}[I_1, +] = \text{closure}(\{E \rightarrow E \cdot +T\})$

$= \{E \rightarrow E \cdot +T, T \rightarrow T \cdot *F, T \rightarrow F \cdot, F \rightarrow (\cdot E), F \rightarrow id\} = I_6$

$goto[I_2, *] = closure(\{T \rightarrow T^*.F\}) = \{T \rightarrow T^*.F, F \rightarrow \lambda(E), F \rightarrow \lambda(id)\} = I_7$
 $goto[I_4, E] = closure(\{F \rightarrow (E.), E \rightarrow E.+T\}) = \{F \rightarrow (E.), E \rightarrow E.+T\} = I_8$
 $goto[I_4, T] = closure(\{E \rightarrow T., T \rightarrow T.*F\}) = \{E \rightarrow T., T \rightarrow T.*F\} = I_2$
 $goto[I_4, F] = closure(\{T \rightarrow F.\}) = \{T \rightarrow F.\} = I_3$
 $goto[I_4, (] = closure(\{F \rightarrow \lambda(E)\}) = I_4$
 $goto[I_4, id] = closure(\{F \rightarrow \lambda(id)\}) = I_5$
 $goto[I_6, T] = closure(\{E \rightarrow E.+T., T \rightarrow T.*F\}) = \{E \rightarrow E.+T., T \rightarrow T.*F\} = I_9$
 $goto[I_6, F] = closure(\{T \rightarrow F.\}) = I_3$
 $goto[I_6, id] = closure(\{F \rightarrow \lambda(id)\}) = I_5$
 $goto[I_7, F] = closure(\{T \rightarrow T^*.F.\}) = \{T \rightarrow T^*.F.\} = I_{10}$
 $goto[I_7, (] = closure(\{F \rightarrow \lambda(E)\}) = I_4$
 $goto[I_7, id] = closure(\{F \rightarrow \lambda(id)\}) = I_5$
 $goto[I_8,)] = closure(\{F \rightarrow (E).)\}) = \{F \rightarrow (E).)\} = I_{11}$
 $goto[I_8, +] = closure(\{E \rightarrow E.+T.\}) = I_6$
 $goto[I_9, *] = closure(\{T \rightarrow T^*.F\}) = \{T \rightarrow T^*.F, F \rightarrow \lambda(E), F \rightarrow \lambda(id)\} = I_7$

The transition diagram of the goto function : DFA



Construction of SLR parsing Table

To construct the action and goto parts of a SLR parsing table, use the following algorithm.

1. Given the grammar G , construct an augmented grammar G' by introducing a production of the form $S' \rightarrow S$ where S is the start symbol of the grammar G .
 2. Let $I_0 = \text{closure}(\{S' \rightarrow .S\})$, starting from I_0 construct SLR (canonical collection of sets of LR(0) items for G' using closure and goto: $C \leftarrow \{I_0, \dots, I_n\}$
 3. State i is constructed from I_i . then the parsing actions for state i are defined as follows:
 - a. If $\text{goto}[I_i, a] = I_j$, set $\text{action}[i, a] = \text{"shift } j\text{"}$, here a must be a terminal.
 - b. If I_i has a production of the form $A \rightarrow \alpha.$, then for all symbol in $\text{FOLLOW}(A)$, set $\text{action}[i, a] = \text{"Reduce } A \rightarrow \alpha\text{"}$, here A should not be S'
 - c. If $[S' \rightarrow S.]$ is in I_i , then set $\text{action}[i, \$] = \text{Accept}$.
 4. If $\text{goto}[I_i, A] = I_j$, where A is non terminal, then set $\text{goto}[i, A] = j$
 5. For all blank entries not defined by step 2 to 3, set error.
 6. The start state s_0 corresponds to $I_0 = \text{closure}(\{S' \rightarrow .S\})$.
- For any conflicting actions are generated by the above rules, we say the grammar is not SLR. The algorithm fails to produce a parser in this case.
 - The parsing table constructed by this algorithm is SLR(1) parsing table of G .
 - An LR parser using this SLR(1) table is called SLR(1) parser and simple called as SLR parser.

Example: Construct the SLR parsing table for the following grammar.

$E \rightarrow E+T \mid T$
 $T \rightarrow T*F \mid F$
 $F \rightarrow (E) \mid id$

Solution: The augmented grammar G' is:

$E' \rightarrow E$
 $E \rightarrow E+T \mid T$
 $T \rightarrow T*F \mid F$
 $F \rightarrow (E) \mid id$

The canonical collection of set of LR(0) items for this grammar are:

$I_0: E' \rightarrow .E$ $E \rightarrow .E+T$ $E \rightarrow .T$ $T \rightarrow .T*F$ $T \rightarrow .F$ $F \rightarrow .(E)$ $F \rightarrow .id$	$I_2: E \rightarrow T.$ $T \rightarrow T.*F$	$I_5: F \rightarrow id.$	$I_8: F \rightarrow (E.)$ $E \rightarrow E.+T$
$I_1: E' \rightarrow E.$ $E \rightarrow E.+T$	$I_3: T \rightarrow F.$	$I_6: E \rightarrow E+.T$ $T \rightarrow T.*F$ $T \rightarrow .F$ $F \rightarrow .(E)$ $F \rightarrow .id$	$I_9: E \rightarrow E+T.$ $T \rightarrow T.*F$
	$I_4: F \rightarrow (.E)$ $E \rightarrow .E+T,$ $E \rightarrow .T,$ $T \rightarrow T.*F$ $T \rightarrow .F$ $F \rightarrow .(E)$ $F \rightarrow .id$	$I_7: T \rightarrow T*.F$ $F \rightarrow .(E)$ $F \rightarrow .id$	$I_{10}: T \rightarrow T*F.$
			$I_{11}: F \rightarrow (E).$

Now computing the entry for the SLR parsing table for action table,

Consider the set of items

For I_0 :

The item $F \rightarrow (E)$ gives to the entry for **action**[0,(] = **shift 4**

The item $F \rightarrow id$ gives to the entry for **action** [0,id] = **shift 5**

The other items in I_0 yield no action. Similarly

For I_1 :

action[1,\$] = **accept** since $E' \rightarrow E$ is in I_1

action[1,+] = **shift 6**

For I_2 :

action[2,*] = **shift 7** since $T \rightarrow T * F$ is in I_2

From $E \rightarrow T$, FOLLOW(E) = { \$,+,)} so **action**[2,\$]=**action**[2,+]=**action**[2,)]=**reduce** $E \rightarrow T$

For I_3 :

$T \rightarrow F$, so FOLLOW(T) = { *,\$,+,)}

So, **action**[3,*] = **action**[3,\$] = **action**[3,+] = **action**[3,)] = **reduce** $T \rightarrow F$

For I_4 :

action[4,(] = **shift 4**

action[4,id] = **shift 5**

For I_5 :

FOLLOW(F) = { *,\$,+,)}

So, **action**[5,*] = **action**[5,\$] = **action**[5,+] = **action**[5,)] = **reduce** $F \rightarrow id$

Similarly,

For I_6 :

action[6,(] = **shift 4**

action[6,id] = **shift 5**

For I_7 :

action[7,(] = **shift 4**

action[7,id] = **shift 5**

For I_8 :

action[8,)] = **shift 11**

action[8,+] = **shift 6**

For I_9 :

$E \rightarrow E + T$, FOLLOW(E) = { \$,+,)}

So, **action**[9,\$]=**action**[9,+]=**action**[9,)]=**reduce** $E \rightarrow E + T$ and

action[9,*] = **shift 7**

For I_{10} :

$T \rightarrow T * F$, FOLLOW(T) = { *,\$,+,)}

So, **action**[10,*] = **action**[10,\$] = **action**[10,+] = **action**[10,)] = **reduce** $T \rightarrow T * F$

For I_{11} :

$F \rightarrow (E)$, FOLLOW(F) = { *,\$,+,)}

So, **action**[11,*] = **action**[11,\$] = **action**[11,+] = **action**[11,)] = **reduce** $F \rightarrow (E)$

Now the action table for SLR parsing for above grammar is:

States	Terminals					
	id	+	*	()	\$
0	shift 5			shift 4		
1		shift 6				accept
2		$E \rightarrow T$			$E \rightarrow T$	$E \rightarrow T$
3		$T \rightarrow F$	$T \rightarrow F$		$T \rightarrow F$	$T \rightarrow F$
4	shift 5			shift 4		
5		$F \rightarrow id$	$F \rightarrow id$		$F \rightarrow id$	$F \rightarrow id$
6	shift 5			shift 4		
7	shift 5			shift 4		
8		shift 6			shift 11	
9		$E \rightarrow E+T$	shift 7		$E \rightarrow E+T$	$E \rightarrow E+T$
10		$T \rightarrow T*F$	$T \rightarrow T*F$		$T \rightarrow T*F$	$T \rightarrow T*F$
11		$F \rightarrow (E)$	$F \rightarrow (E)$		$F \rightarrow (E)$	$F \rightarrow (E)$

Now goto table for SLR.

States→ Variable↓	E	T	F
0	1	2	3
1			
2			
3			
4	8	2	3
5			
6		9	3
7			10
8			
9			
10			
11			

Using the above parsing table, the SLR Parser takes the following moves to parse the string *id*id+id*

<u>Stack</u>	<u>Input</u>	<u>Action</u>	<u>Output</u>
0	id*id+id\$	shift 5	
0id5	*id+id\$	reduce by $F \rightarrow id$	$F \rightarrow id$
0F3	*id+id\$	reduce by $T \rightarrow F$	$T \rightarrow F$
0T2	*id+id\$	shift 7	
0T2*7	id+id\$	shift 5	
0T2*7id5	+id\$	reduce by $F \rightarrow id$	$F \rightarrow id$
0T2*7F10	+id\$	reduce by $T \rightarrow T * F$	$T \rightarrow T * F$
0T2	+id\$	reduce by $E \rightarrow T$	$E \rightarrow T$
0E1	+id\$	shift 6	
0E1+6	id\$	shift 5	
0E1+6id5	\$	reduce by $F \rightarrow id$	$F \rightarrow id$
0E1+6F3	\$	reduce by $T \rightarrow F$	$T \rightarrow F$
0E1+6T9	\$	reduce by $E \rightarrow E + T$	$E \rightarrow E + T$
0E1	\$	accept	

Properties of SLR grammars

Every SLR(1) grammar is unambiguous. But there exist certain unambiguous grammar that are not SLR(1). In such a grammars there exists at least one multiply defined entry action[i,a] which contains both shift directive and reduce directive.

Invalid reduction:

In SLR parser , in any state i, a reduction $A \rightarrow \alpha$ is performed on input symbol 'a' if state i contains $[A \rightarrow \alpha.]$ and 'a' is in FOLLOW(A), however not all symbols in FOLLOW(A) can be reduced in such a fashion.

For example, consider following grammar.

G: $S \rightarrow L = R$ $S \rightarrow R$ $L \rightarrow *R$ $L \rightarrow id$ $R \rightarrow L$	\Rightarrow	G' : $S' \rightarrow S$ $S \rightarrow L = R$ $S \rightarrow R$ $L \rightarrow *R$ $L \rightarrow id$ $R \rightarrow L$
--	---------------	--

The canonical collection of the item sets of LR(0) for this grammar are

I₀: $S' \rightarrow . S$ $S \rightarrow . L = R$ $S \rightarrow . R$ $L \rightarrow . * R$ $L \rightarrow . id$ $R \rightarrow . L$	I₁: $S' \rightarrow S .$ I₂: $S \rightarrow L . = R$ $R \rightarrow L .$ I₃: $S \rightarrow R .$	I₄: $L \rightarrow * . R$ $R \rightarrow . L$ $L \rightarrow . * R$ $L \rightarrow . id$	I₅: $L \rightarrow id .$ I₆: $S \rightarrow L = . R$ $R \rightarrow . L$ $L \rightarrow . * R$ $L \rightarrow . id$	I₇: $L \rightarrow * R .$ I₈: $R \rightarrow L .$ I₉: $S \rightarrow L = R .$
--	---	--	---	---

Now: while construction the parsing table, consider the item set I_2 :

$\text{goto}[I_2, =] = I_6$ this will make entry in SLR parsing table as $\text{action}[2, =] = \text{shift } 6$

also $R \rightarrow L$ is also in I_2 , so by the rule of constructing SLR parsing table, we compute $\text{FOLLOW}(R) = \{ =, \$ \}$ this will make entry to SLR parsing table as $\text{action}[2, =] = \text{reduce } R \rightarrow L$.

So the entry on action table for $\text{action}[2, =]$ is multiply defined, one is shift operation and another is reduce operation. This leads the SLR parser into shift-reduce conflict for the state 2 and input '='.

The grammar is not ambiguous but there is shift-reduce conflict. So the SLR parser is not enough powerful to remember left context to decide what action the parser should take on input '='.

LR(k) items

In order to avoid invalid reductions, the general form of an item is of the form $[A \rightarrow \alpha.\beta, a]$ i.e. extra information is put into a state by including a terminal symbol as a second component in an item. In this case, the second term a has no effect when β is not empty, but in an item of the form $[A \rightarrow \alpha., a]$, reduce action is performed using form $[A \rightarrow \alpha]$ only if the next input symbol is ' a '.

Such an item is called a LR(1) item where the input symbol ' a ' is called the "lookahead" whose length is 1.

The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that *closure* and *goto* operations work a little bit different.

Computation of closure(1) and LR(1) items.

1. Repeat
2. For each item of the form $[A \rightarrow \alpha.B\beta, a]$ in I ,
each production of the form $B \rightarrow \gamma$ in G' and
each terminal ' b ' in $\text{FIRST}(\beta a)$ do
add $[B \rightarrow \gamma.b]$ to I if it is not already there.
3. Until no more items can be added to I .

Computation of goto[I,X] for LR(1) items

Given the set of all items of the form $[A \rightarrow \alpha.X\beta, a]$ in I ,
 $\text{goto}[I, X] = \text{closure}(\{A \rightarrow \alpha.X.\beta, a\})$.

Computation of LR(1) DFA

1. Start with $C = \{\text{closure}(\{S' \rightarrow .S, \$\})\}$ where S is start symbol.
2. Repeat
3. For each set of items I in C and each grammar symbol X such that $\text{goto}[I, X]$ is not already in C , do,
Add $\text{goto}[I, X]$ to C .
4. Until no more sets of items can be added to C .

Example: Compute the LR(1) collection of items from the following grammar.

$S \rightarrow CC$

$C \rightarrow cC / d$

Solution: The augmented grammar is

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC / d$

$I_0: \text{closure}(\{(S' \rightarrow \bullet S, \$)\})$ $= \{ (S' \rightarrow \bullet S, \$)$ $(S \rightarrow \bullet C C, \$)$ $(C \rightarrow \bullet c C, c/d)$ $(C \rightarrow \bullet d, c/d) \}$	$I_4: \text{goto}(I_0, d) =$ $(C \rightarrow d \bullet, c/d)$	$\text{goto}(I_3, c) = I_3$ $\text{goto}(I_3, d) = I_4$
$I_1: \text{goto}(I_0, S)$ $= (S' \rightarrow S \bullet, \$)$	$I_5: \text{goto}(I_2, C) =$ $(S \rightarrow C C \bullet, \$)$	$I_9: \text{goto}(I_6, C) =$ $(C \rightarrow c C \bullet, \$)$
$I_2: \text{goto}(I_0, C) =$ $(S \rightarrow C \bullet C, \$)$ $(C \rightarrow \bullet c C, \$)$ $(C \rightarrow \bullet d, \$)$	$I_6: \text{goto}(I_2, c) =$ $(C \rightarrow c \bullet C, \$)$ $(C \rightarrow \bullet c C, \$)$ $(C \rightarrow \bullet d, \$)$	$\text{goto}(I_6, c) = I_6$ $: \text{goto}(I_6, d) = I_7$
$I_3: \text{goto}(I_0, c) =$ $(C \rightarrow c \bullet C, c/d)$ $(C \rightarrow \bullet c C, c/d)$ $(C \rightarrow \bullet d, c/d)$	$I_7: \text{goto}(I_2, d) =$ $(C \rightarrow d \bullet, \$)$	
	$I_8: \text{goto}(I_3, C) =$ $(C \rightarrow c C \bullet, c/d)$	

Note: In first case $[S' \rightarrow \bullet S, \$]$ is of the form $[A \rightarrow \alpha B \beta, a]$ where β is empty and $a = \$$. It has to be added the item $[B \rightarrow \bullet, b]$ for each terminal 'b' in $\text{FIRST}(\beta a)$ which is equal to '\$'

So add $S \rightarrow \bullet C C, \$)$

Similarly for case $S \rightarrow \bullet C C, \$)$, $(\beta a) = (C\$)$ and $\text{FIRST}(C\$) = \{c, d\}$ so it has to be added the items

$C \rightarrow \bullet c C, c/d$

$C \rightarrow \bullet d, c/d$

The Core of LR(1) Items

The core of a set of LR(1) Items is the set of their first components (i.e., LR(0) items). For example the core of the set of LR(1) items

$\{ (C \rightarrow c \bullet C, c/d),$
 $(C \rightarrow \bullet c C, c/d),$
 $(C \rightarrow \bullet d, c/d) \}$

is

$\{ C \rightarrow c \bullet C,$
 $C \rightarrow \bullet c C,$
 $C \rightarrow \bullet d$
 $\}$

Construction of the LR(1) parsing table

To construct the action and goto parts of a LR(1) parsing table, use the following algorithm.

1. Given the grammar G , construct an augmented grammar G' by introducing a production of the form $S' \rightarrow S$ where S is the start symbol of the grammar G .
2. Let $I_0 = \text{closure}(\{S' \rightarrow .S, \$\})$, starting from I_0 construct LR(1) (canonical collection of sets of LR(1) items for G' using closure and goto: $C \leftarrow \{I_0, \dots, I_n\}$
3. State i is constructed from I_i . then the parsing actions for state i are defined as follows:
 - a. If $\text{goto}[I_i, a] = I_j$, set $\text{action}[i, a] = \text{"shift } j\text{"}$, here a must be a terminal.
 - b. If I_i has a production of the form $[A \rightarrow \alpha. a]$, set $\text{action}[i, a] = \text{"Reduce } A \rightarrow \alpha\text{"}$, here A should not be S'
 - c. If $[S' \rightarrow S., \$]$ is in I_i , then set $\text{action}[i, \$] = \text{Accept}$.
4. If $\text{goto}[I_i, A] = I_j$, where A is non terminal, then set $\text{goto}[i, A] = j$
5. For all blank entries not defined by step 2 to 3, set error.
6. The start state s_0 corresponds to $I_0 = \text{closure}(\{S' \rightarrow .S, \$\})$.

Now the parsing table for the grammar given above as

1. $S' \rightarrow S$
2. $S \rightarrow CC$
3. $C \rightarrow cC$
4. $C \rightarrow d$

will be as:

Terminals/ States	action			goto	
	c	d	\$	S	C
0	s3	s4		1	2
1			accept		
2	s6	s7			5
3	s3	s4			8
4	R3	R3			
5			R1		
6	s6	s7			9
7			R3		
8	R2	R2			
9			R2		

Here **R2** means Reduce by Production 2 of grammar above

i.e. Reduce $S \rightarrow CC$ and so on.

s_i means shift i i.e. s_3 refers as "shift 3"

LALR Grammar

LALR(Lookahead LR) grammars are midway in complexity between SLR and canonical LR(most complex) grammars. They perform a “generalization” over canonical LR itemsets.

A typical programming language generates thousands of states for canonical LR parser while they generate only hundreds of states for SLR and LALR. So it is much easier and economical to construct the SLR and LALR parser.

Union operation on items.

- Given an item of the form $[A \rightarrow \alpha B \beta, a]$, the first part of the item $A \rightarrow \alpha B \beta$ is called the “core” of the item.
- Given two states of the form $I_i = A \rightarrow \alpha, a$ and $I_j = A \rightarrow \alpha, b$ the core of I_i and I_j is same only the difference is the second part of the item. So the union of these two items is defined as

$$I_{ij} = \{[A \rightarrow \alpha, a/b]\}.$$
- The I_{ij} will perform the reduce operation on seeing either ‘a’ or ‘b’ on input buffer. The state machine resulting from the above union operation has one less state.
- If I_i and I_j have more than one items, then the set of all core elements in I_i should be the same as the set of all core elements in I_j for the union operation to be possible.
- Union operation does not create any new shift-reduce conflicts but can create new reduce-reduce conflicts. Shift operation only depends on the core and not on the next input symbol.

Construction of the LALR parsing table

To construct the action and goto parts of a LALR parsing table, use the following algorithm.

1. Given the grammar G , construct an augmented grammar G' by introducing a production of the form $S' \rightarrow S$ where S is the start symbol of the grammar G .
2. For each core present; find all sets having that same core; replace those sets having same cores with a single set which is their union. $C = \{I_0, \dots, I_n\} \rightarrow C' = \{J_1, \dots, J_m\}$ where $m \leq n$
3. Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.
 If $J = I_1 \cup \dots \cup I_k$ since I_1, \dots, I_k have same cores
 \rightarrow cores of $\text{goto}(I_1, X), \dots, \text{goto}(I_k, X)$ must be same.
4. So, $\text{goto}(J, X) = K$ where K is the union of all sets of items having same cores as $\text{goto}(I_1, X)$.

If no conflict is introduced, the grammar is LALR(1) grammar. The above algorithm is inefficient since it constructs the entire canonical DFA before generating the LALR parsing table.

Example:

Compute the LR(1) collection of items from the following grammar.

$S \rightarrow CC$

$C \rightarrow xC / d$

Solution: The augmented grammar is

$$S' \rightarrow S$$

$$S \rightarrow CC$$

$$C \rightarrow cC / d$$

The canonical LR(1) items computed from this grammar are $\{ I_0, I_1, \dots, I_n \}$ as computed in previous LR(1) parsing table.

In the collection of LR(1) items, I_3 and I_6 , I_4 and I_7 , I_8 and I_9 have same core items respectively. So performing union operations, the items for LALR will be as

$I_0: \text{closure}(\{(S' \rightarrow \bullet S, \$)\})$ $= \{ (S' \rightarrow \bullet S, \$)$ $(S \rightarrow \bullet C C, \$)$ $(C \rightarrow \bullet c C, c/d)$ $(C \rightarrow \bullet d, c/d) \}$	$I_2: \text{goto}(I_0, C) =$ $(S \rightarrow C \bullet C, \$)$ $(C \rightarrow \bullet c C, \$)$ $(C \rightarrow \bullet d, \$)$	$I_{47}: \text{goto}(I_0, d) =$ $(C \rightarrow d \bullet, c/d/\$)$
$I_1: \text{goto}(I_0, S)$ $= (S' \rightarrow S \bullet, \$)$	$I_{36}: \text{goto}(I_0, c) =$ $(C \rightarrow c \bullet C, c/d/\$)$ $(C \rightarrow \bullet c C, c/d/\$)$ $(C \rightarrow \bullet d, c/d/\$)$	$I_5: \text{goto}(I_2, C) =$ $(S \rightarrow C C \bullet, \$)$
		$I_{89}: \text{goto}(I_3, C) =$ $(C \rightarrow c C \bullet, c/d/\$)$

Now,

$\text{goto}[I_{36}, C] = I_{89}$, since $\text{goto}[I_3, C] = I_8$ in original set of LR(1) items. Similarly $\text{goto}[I_2, c] = I_{36}$, since $\text{goto}[I_2, c] = I_6$ in original LR(1) items and so on.

So the LALR parsing table for the grammar has 3 less states as

Terminals/ States	action			goto	
	c	d	\$	S	C
0	s36	s4		1	2
1			accept		
2	s36	s7			5
36	s36	s4			89
47	R3	R3	R3		
5			R1		
89	R2	R2	R2		

Now for input ccd the parser takes the following steps in LR(1) parser

Stack	inputbuff	action	output
\$0	ccd\$	shift 3	
\$0c3	cd\$	shift 3	
\$0c3c3	d\$	shift 4	
\$0c3c3d4	\$	Error	

In LALR using above table,

Stack	inputbuff	action	output
\$0	ccd\$	shift 36	
\$0c36	cd\$	shift 36	
\$0c36c36	d\$	shift 47	
\$0c36c36d47	\$	reduce $C \rightarrow d$	$C \rightarrow d$
\$0c36c36C89	\$	reduce $C \rightarrow cC$	$C \rightarrow cC$
\$0c36C89	\$	reduce $C \rightarrow cC$	$C \rightarrow cC$
\$0C2	\$	Error	

So for both LR(1) and LALR parser leads to an Error for string **ccd**.

Now consider for string *cdd*

LR(1) parser				LALR Parser			
Stack	Input	Action	output	Stack	Input	Action	output
\$0	cdd\$	shift 3		\$0	cdd\$	shift 36	
\$0c3	dd\$	shift 4		\$0c36	dd\$	shift 47	
\$0c3d4	d\$	Red. $C \rightarrow d$	$C \rightarrow d$	\$0c36d47	d\$	Red. $C \rightarrow d$	$C \rightarrow d$
\$0c3C8	d\$	Red. $C \rightarrow cC$	$C \rightarrow cC$	\$0c36C89	d\$	Red. $C \rightarrow cC$	$C \rightarrow cC$
\$0C2	d\$	shift 7		\$0C2	d\$	shift 47	
\$0C2d7	\$	Red. $C \rightarrow d$	$C \rightarrow d$	\$0C2d47	\$	Red. $C \rightarrow d$	$C \rightarrow d$
\$0C2C	\$	Red. $S \rightarrow CC$	$S \rightarrow CC$	\$0C2C5	\$	Red. $S \rightarrow CC$	$S \rightarrow CC$
\$0S1	\$	Acctpt		\$0S1	\$	Accept	

For the parsing of valid string of grammar we saw that both LR(1) and LALR take same actions for parsing. But for invalid string like ccd, the LR(1) parser leads to error after some shift action where LALR proceeds some reductions after LR parser has detected an error. But finally, LALR also discovered the error.

Kernel and non-kernel items

In order to devise a more efficient way of building LALR parsing tables, we define kernel and non-kernel items. Those items that are either the initial item $[S' \rightarrow .S, \$]$ or the items that have somewhere dot(.) other than the beginning of the right side are called the kernel items. No item generated by a goto has dot at the left end of production. Items that are generated by closure over kernel items hav a dot at the beginning of production. These items are called non-kernel items.

Chapter 5: Syntax Directed Translation

In any programming language, grammar symbols are associated with attributes to associate information with the language construct that they represent. An attribute may hold almost any thing - a string, a number, a memory location, a complex record.

The values of these attributes are evaluated by the semantic rules associated with the production of the grammar. The evaluation of these semantic rules may generate intermediate code, put information on the symbol table, perform type checking, issue error message etc. When we associate semantic rules with productions, we use two notations:

- **Syntax-Directed Definitions**
- **Translation Schemes**

Syntax directed definitions: A syntax directed definitions are high level specification for translations. They hide many implementation details such as the order in which translation takes place. We associate a production rule with a set of semantic actions and we do not say when they will be evaluated.

Translation Schemes: The translation schemes indicates the order in which semantic rules are to be evaluated. In other words, translation schemes give a little bit information about implementation details. using dependency graph. So they allow some implementation details to be shown.

A syntax-directed definition is a generalization of a context-free grammar in which:

- Each grammar symbol is associated with a set of attributes.
- This set of attributes for a grammar symbol is partitioned into two subsets called
 - **synthesized** and
 - **inherited** attributes of that grammar symbol.
- Each production rule is associated with a set of semantic rules.
- *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*.

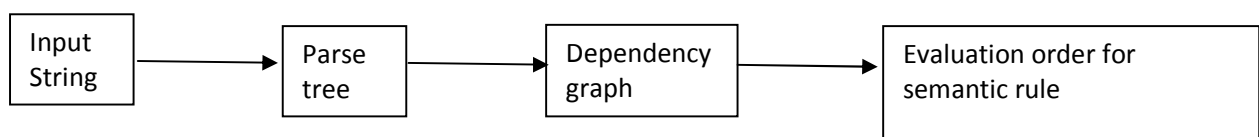
This *dependency graph* determines the evaluation order of these semantic rules. Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

The value of synthesized attributes of a node are determined by the children of the node where as the value of inherited attributes of node are determined by the parent and siblings of the node. To determine the attributes of the nodes in parse tree, we annotate the parse tree.

Annotated Parse Tree:

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.
- The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.
- Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

The conceptual view of syntax directed translation is as:



In syntax directed definition, each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form $b = f(c_1, c_2, c_3 \dots c_n)$ where f is a function and b can be one of the followings:

1. b is synthesized attribute of A and $c_1, c_2, c_3 \dots c_n$ are attributes of the grammar symbols of the production ($A \rightarrow \alpha$)
2. b is an inherited attribute of one of the grammar symbols on the right side of the production and $c_1, c_2, c_3 \dots c_n$ are attributes of the grammar symbols in production ($A \rightarrow \alpha$)

Attribute Grammar:

- So, a semantic rule $b = f(c_1, c_2, \dots, c_n)$ indicates that the attribute b *depends on* attributes c_1, c_2, \dots, c_n .
- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.
- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

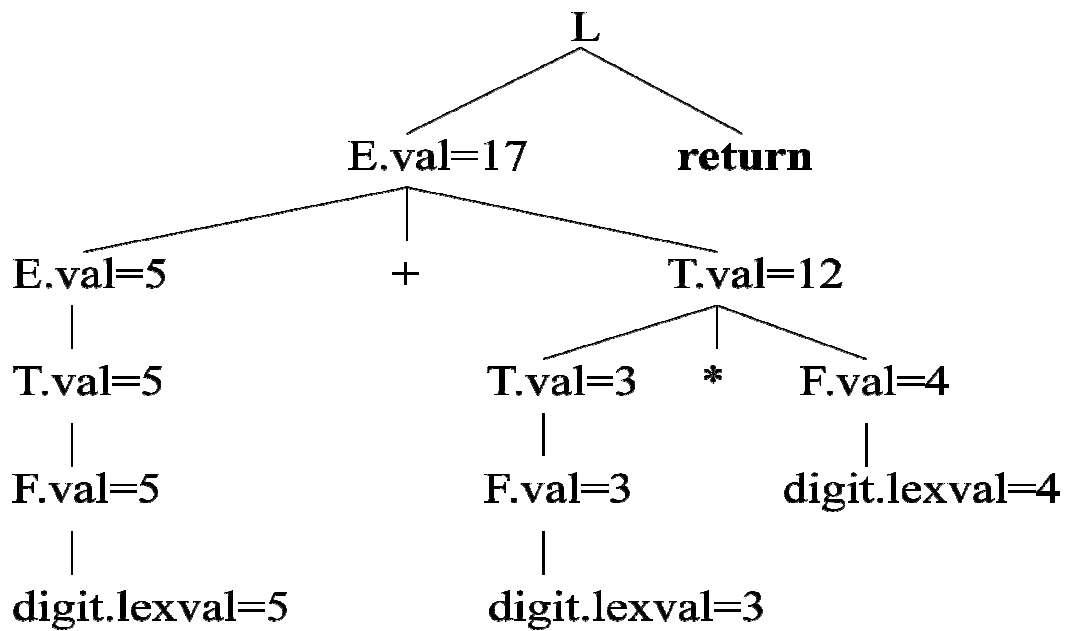
Functions in semantic rules will often be written as expressions. Below is the example of an syntax directed definitions

<u>Production</u>	<u>Semantic Rules</u>
$L \rightarrow E \text{ return}$	$\text{print}(E.\text{val})$
$E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$
$E \rightarrow T$	$E.\text{val} = T.\text{val}$
$T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$
$T \rightarrow F$	$T.\text{val} = F.\text{val}$
$F \rightarrow (E)$	$F.\text{val} = E.\text{val}$
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.\text{lexval}$

- Here, Symbols E , T , and F are associated with a synthesized attribute *val*.
- The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).
- In STD, terminals are assumed to have synthesized attributes only and they usually supplied by the lexical analyzer.

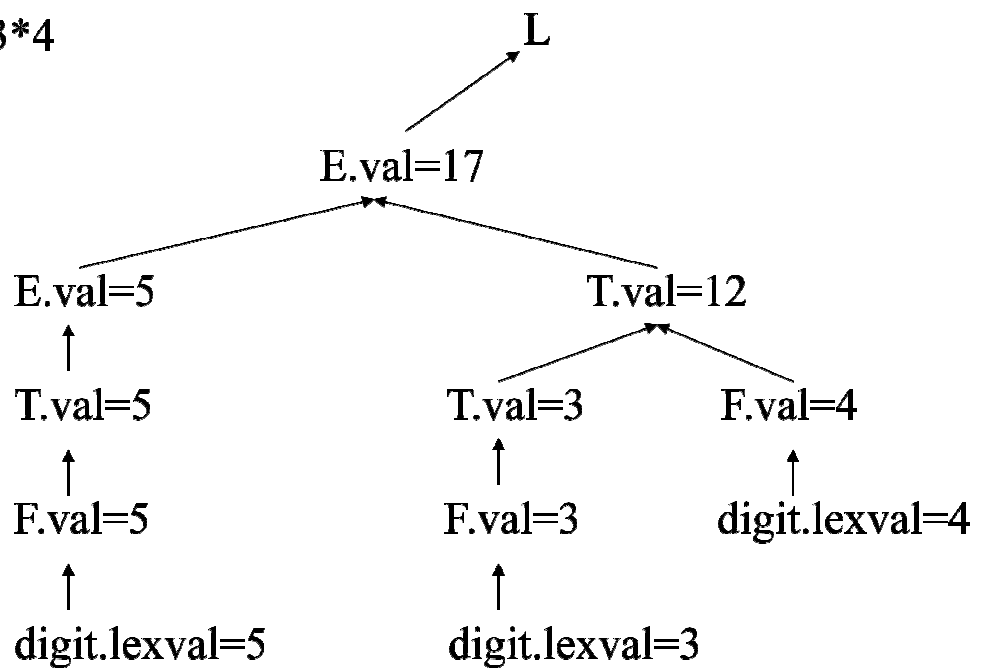
S-attributed definition:

A syntax directed definition that uses synthesized attributes exclusively is said to be an **S-attributed definition**. A parse tree for S-attributed definition can always be annotated by evaluating the semantic rules for attributes at each node in bottom up manner. The evaluation of s-attributed definition is based on the Depth first Traversal of the annotated tree.



The parse tree with annotation for an input expression of the grammar as : **5+3*4** ret is,

Input: 5+3*4



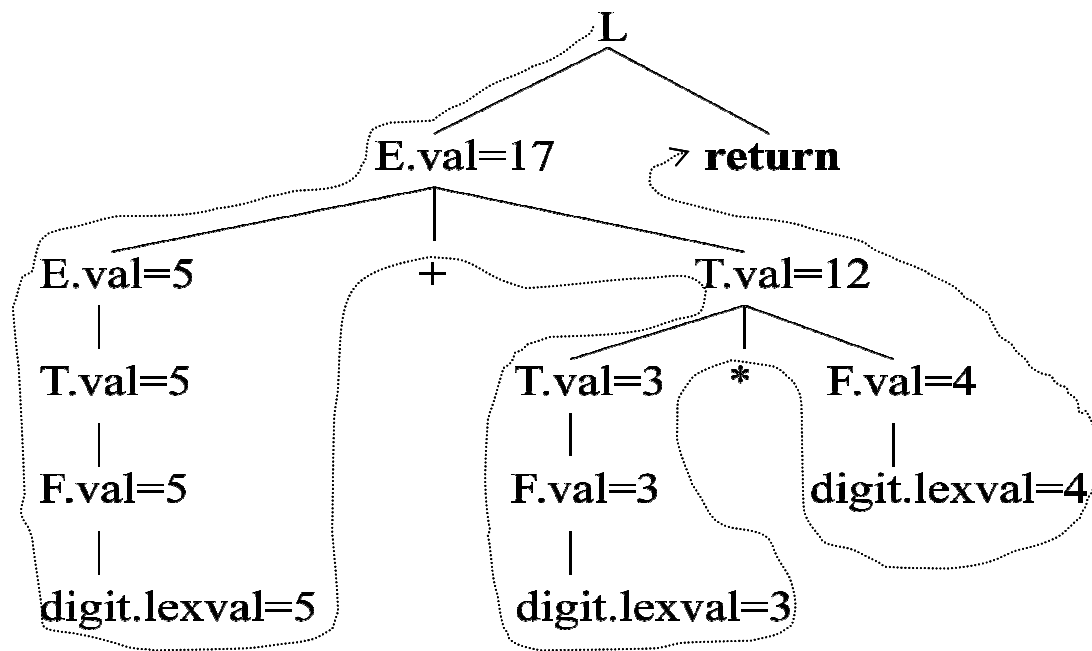


Figure: The DFT of annotated graph

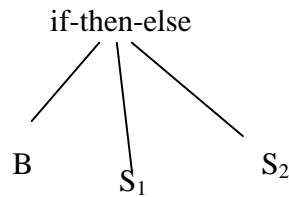
Production**Semantic Rules** $E \rightarrow E_1 + T$ $E.loc = \text{newtemp}(), E.code = E_1.code \parallel T.code \parallel \text{add } E_1.loc, T.loc, E.loc$ $E \rightarrow T$ $E.loc = T.loc, E.code = T.code$ $T \rightarrow T_1 * F$ $T.loc = \text{newtemp}(), T.code = T_1.code \parallel F.code \parallel \text{mult } T_1.loc, F.loc, T.loc$ $T \rightarrow F$ $T.loc = F.loc, T.code = F.code$ $F \rightarrow (E)$ $F.loc = E.loc, F.code = E.code$ $F \rightarrow \text{id}$ $F.loc = \text{id.name}, F.code = ""$

- Symbols E, T, and F are associated with synthesized attributes *loc* and *code*.
- The token **id** has a synthesized attribute *name* (it is assumed that it is evaluated by the lexical analyzer).
- It is assumed that \parallel is the string concatenation operator.

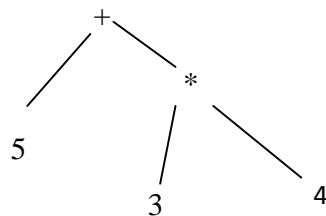
Construction of syntax Tree:

The syntax directed definitions can be used to specify the construction of syntax trees and other graphical representation. The syntax tree is an intermediate representation of the expression.

A syntax tree is a condensed form of parse tree useful for representing language constructs. The production $S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$ might appear in syntax tree as



In a syntax tree, operators and keywords do not appear as leaves and they appear in interior nodes that will be the parent of those leaves in the parse tree. Following is the syntax tree for the expression $5+3*4$ will be as



Syntax directed translation can be based on syntax trees as well as parse trees. The approach is same for both case. We attach attributes to the nodes as in parse tree. For construct the syntax tree for the language construct, the syntax directed definitions should be defined to create the nodes, assign attributes and link those node to other nodes.

Construction of the syntax tree is similar to the translation of the expression into postfix form. The sub-tree is constructed for each sub-expression by creating a node for each operator and operand. The children of an operator node are the

Each node in a syntax tree is a record with several necessary fields. If a node is operator then it has the pointer field to point its children(operands). So there are two types of nodes : interior nodes (operator nodes) and leaf node.

Consider the following grammar with associated semantic rules:

$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow E_1 - T$	$E.val = E_1.val - T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow (E)$	$T.val = E.val$
$T \rightarrow \text{id}$	$T.val = \text{id}.entry$
$T \rightarrow \text{num}$	$T.val = \text{num}.val$

The syntax directed definition to create the parse tree for the expression must execute the implementation of the following functions.

1. *makenode(op,left,right)*: To create an operator node with label op and two fields containing pointer left and right.
2. *makeleaf(id,entry)*: creates an identifier node with label id and a field containing entry, a pointer to the symbol table entry for that id.
3. *makeleaf(num,val)*: creates a node with label num and a field containing val, the value of the number.

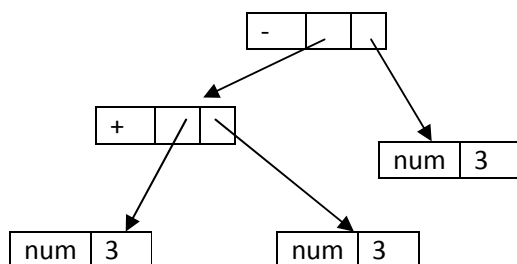
The syntax directed definition for construction of syntax tree using these semantic rule will be as:

Production	Semantic rules
$E \rightarrow E_1 + T$	$E.ptr = makenode('+', E_1.ptr, T.ptr)$
$E \rightarrow E_1 - T$	$E.ptr = makenode('-', E_1.ptr, T.ptr)$
$E \rightarrow T$	$E.ptr = T.ptr$
$T \rightarrow (E)$	$T.ptr = E.ptr$
$T \rightarrow id$	$T.ptr = makeleaf(id, id.entry)$
$T \rightarrow num$	$T.ptr = makeleaf(num, num.val)$

So for expression 3+5-4, sequence of execution of the following fragments will create the syntax tree

1. $p_1 = makeleaf(num, 3);$
2. $p_2 = makeleaf(num, 5);$
3. $p_3 = makeleaf(num, 4);$
4. $p_4 = makenode('+', p_1, p_2);$
5. $p_5 = makenode('-', p_4, p_3);$

The syntax tree created above will be as:



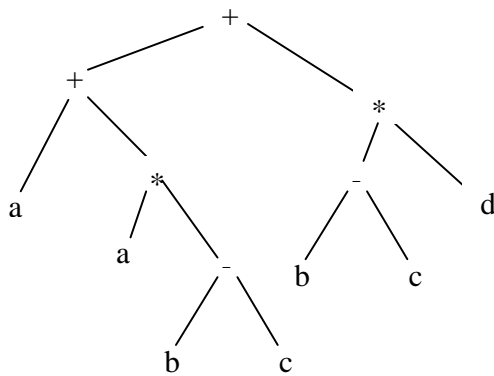
Directed Acyclic graphs for expression:

A directed acyclic graph(DAG) for an expression identifies the common sub-expressions in the expression. Like a syntax tree, a dag has a node for every sub-expression of the expression. An interior node represents operator and its children represent the operands. The only difference between syntax tree and DAG is that a node representing common sub-expression has more than one parent in the syntax tree.

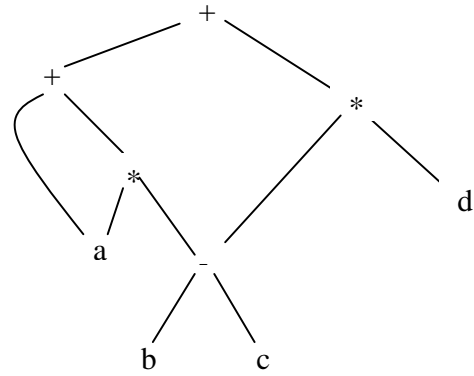
For example, consider the following expression:

$$a + a * (b - c) + (b - c) * d$$

The syntax tree and DAG is as shown below:



Syntax Tree



Directed Acyclic Graph

The sequence of instruction for creating the DAG of above expression using the makenode and makeleaf is as below:

- | | |
|--|---|
| 1. $p1 = \text{makeleaf}(\text{id}, a);$ | 6. $p6 = \text{makenode}('*', p1, p5);$ |
| 2. $p2 = \text{makeleaf}(\text{id}, b);$ | 7. $p7 = \text{makenode}('+', p1, p6);$ |
| 3. $p3 = \text{makeleaf}(\text{id}, c);$ | 8. $p8 = \text{makenode}('*', p5, p4);$ |
| 4. $p4 = \text{makeleaf}(\text{id}, d);$ | 9. $p9 = \text{makenode}('+', p7, p8);$ |
| 5. $p5 = \text{makenode}('-', p2, p3);$ | |

Inherited Attributes:

An inherited attribute at any node is defined based on the attributes at the parent and/or siblings of the nodes. Inherited attributes are useful for describing context sensitive behavior of grammar symbol. For example, an inherited attribute can be used to keep track of whether an identifier appears at the left or right side of an assignment operator. This can be used to decide whether to use the value of the identifier or its l-value.

Example of inherited attributes:

The following example shows the distribution of type information to the various identifiers in a declaration. A declaration generated by a non-terminal D in syntax directed definition consists of keywords int or real followed by list of identifiers 'L'

The non terminal T has a synthesized attribute type whose value is determined by the keyword in declaration.

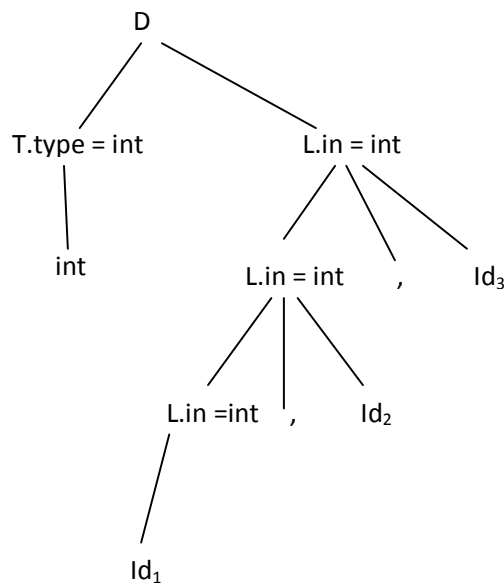
The syntax directed definition for the declaration is:

$D \rightarrow TL \quad \{L.in = T.type\}$
 $T \rightarrow int \quad \{ T.type = integer \}$
 $T \rightarrow real \quad \{ T.type = real \}$
 $L \rightarrow L_1, id \quad \{ L_1.in = L.in \}$
 $\quad \quad \quad \{ addtype(id.entry, L.in) \}$
 $L \rightarrow id \quad \{ addtype(id.entry, L.in) \}$

The semantic rule $L.in = T.type$, associated with the production $D \rightarrow TL$ sets inherited attribute $L.in$ to the type in declaration. Then the rules pass this type down the parse tree using the inherited attribute $L.in$.

The rules associated with the production for L call procedure $addtype()$ to add the type of each identifier to its entry in the symbol table(pointed by attribute entry).

The annotated parse tree for the input string : *int id₁,id₂,id₃* is as shown below.



- The above annotated parse tree shows the inherited attribute $L.in$ at each node labeled L .
- At first level of tree, attribute type is inherited from the sibling $T.type$ where int is synthesized attribute of node labeled $T.type$.
- In another level, the nodes labeled as $L.in$ has the attributes inherited from the parent node(top-down)
- At each L nodes we also call procedure $addtype()$ to insert into the symbol table for the fact that the identifier at the right child of this L -node has type int .

The dependency graph:

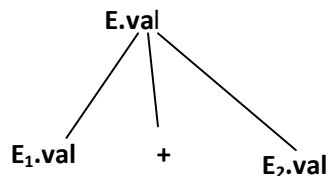
In order to correctly evaluate attributes of syntax tree nodes, a dependency graph is a useful tool. If an attribute b of a node in a parse tree depends on an attribute c , then the semantic rule for b at that node must be evaluated after the semantic rule that defines c . A dependency

graph is a directed graph that contains attributes as nodes and dependencies across attributes as edges.

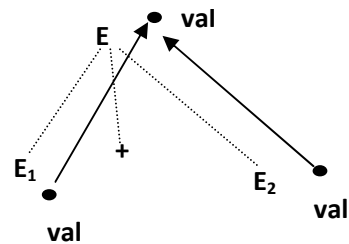
For example

$$E \rightarrow E_1 + E_2 \quad \{ E.val = E_1.val + E_2.val \}$$

The parse Tree is:



The Dependency Graph



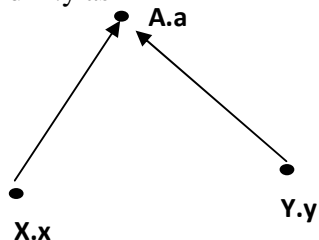
Algorithm for dependency graph

```

For each node n in the parse tree do begin
    For each attribute 'a' of the grammar symbol n do begin
        Construct a node in the dependency graph for a.
    End
End

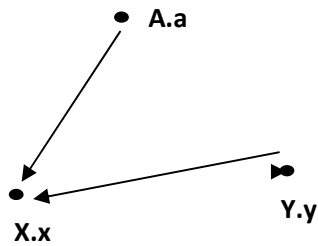
For each node n in parse tree do begin
    For each semantic rule of the form b = f(c1, c2, c3 .... cn) associated with the
    production at n do begin
        For i = 1 to n do begin
            Construct an edge from ci to b
        End
    End
End
  
```

e.g. Suppose $A.a = f(X.x, Y.y)$ is a semantic rule for the production $X \rightarrow XY$ which defines synthesized attribute $A.a$ that depends on the attributes $X.x$ and $Y.y$. For the parse tree for this production, there will be 3 nodes $A.a$, $X.x$, $Y.y$ in the dependency graph with an edge to $A.a$ from both $X.x$ and $Y.y$ as



Dependency graph for $A.a = f(X.x, Y.y)$

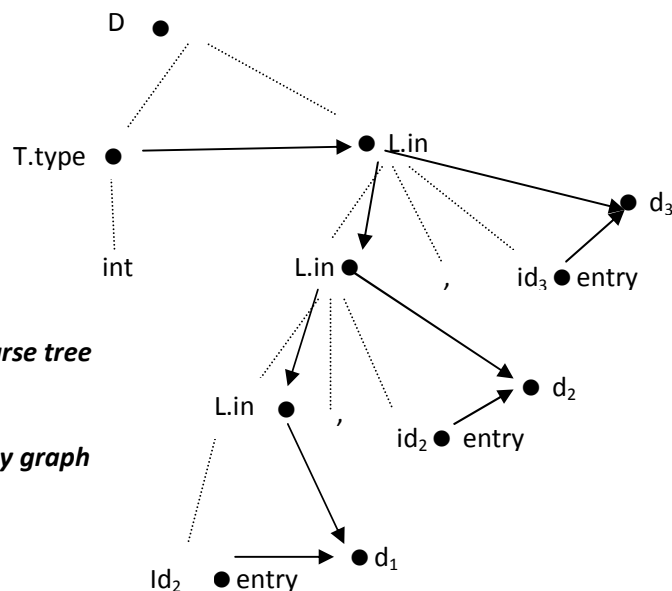
If the production $A \rightarrow XY$ has semantic rule $X.x = g(A.a, Y.y)$ then there will be edge from $A.a$ and $Y.y$ to $X.x$ since $X.x$ depends on $Y.y$ and $A.a$ as:



Dependency graph for $X.x = g(A.a, Y.y)$

For the grammar for declaration, the semantic rule applies in dependency graph for string

Int id₁, id₂, id₃



Here Dotted line shows the parse tree

*Directed solid line shows the
dependency hence dependency graph*

Dependency in above graph shows:

- $L.in = T.type$, $L.in = L.in$
- For semantic rule $addtype(id.entry, L.in)$ associated with the L-Production, leads to creation of a dummy attribute (d_1, d_2, d_3 in graph)

Exercise: A grammar for declaration is given as

$D \rightarrow id L$

$L \rightarrow ,id L | T$

$T \rightarrow integer | real$

Construct a syntax directed definitions to enter the type of each identifier into symbol table using synthesized and inherited attributes. Also construct annotated parse tree for input string as :

id₁, id₂, id₃: integer and construct the dependency graph for the same.

L -attributed definitions:

A syntax directed definition is called L-attributed if in the semantic rule of a production

$A \rightarrow X_1 X_2 X_3 \dots X_n$, the inherited attribute of X_j depends only on:

- The attributes (synthesis or inherited) of the symbols $X_1, X_2, X_3, \dots, X_{j-1}$
- The inherited attributes of A.

An inherited attribute can be evaluated in a left to right fashion using a depth first evaluation order.

Example of L -attributed definition.**Production**

$A \rightarrow XYZ \quad \{X.i = f_1(A.i), Y.i = f_2(Y.s)\}$

$X \rightarrow YZ \quad \{Z.i = f(X.i, Y.i)\}$

Example of syntax directed definition which is not L-attributed.

$A \rightarrow LM \quad \{L.i = f_1(A.i), M.i = f_2(L.s), A.s = f_3(M.s)\}$

$A \rightarrow QR \quad \{R.i = f_4(A.i), Q.i = f_5(R.s), A.s = f_6(Q.s)\}$

Here semantic rule $A.s = f_3(M.s)$ violates the rule for L-attributed definition since $A.s$ depends on the synthesized attribute of M which is on the right side of the production $A \rightarrow LM$ and in semantic rule $A.s = f_6(Q.s)$ also violates L -attributed definition

Translation Schemes: A CFG with “semantic actions” embedded into its productions is a translation scheme. It is useful for binding order or evaluation into parse tree.

For Example: Translation scheme for translation simple infix expression involving

$expr \rightarrow expr + term \{ print(“+”) \}$

$expr \rightarrow expr - term \{ print(“-”) \}$

$expr \rightarrow term$

$term \rightarrow 0 \{ print(“0”) \}$

$term \rightarrow 1 \{ print(“1”) \}$

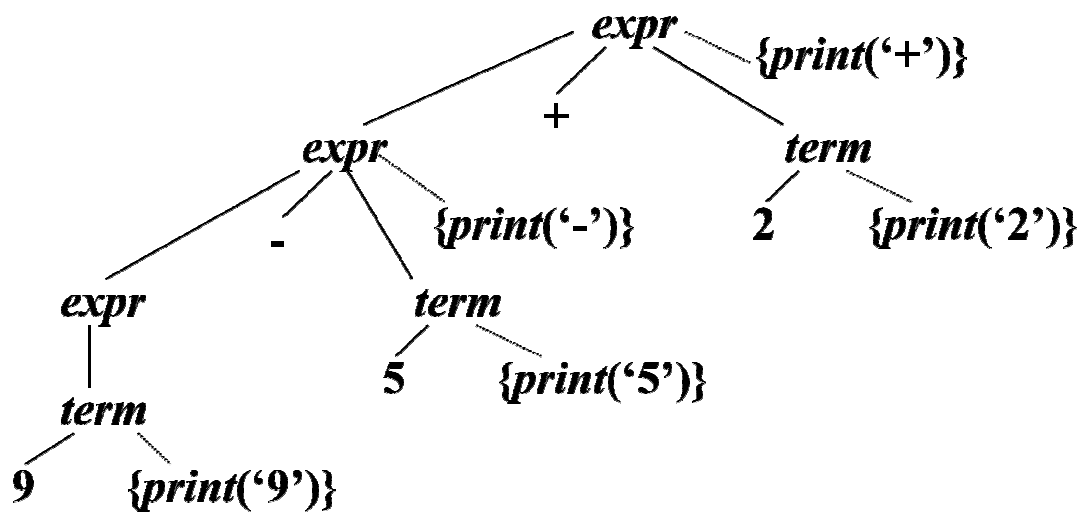
.

.

.

$term \rightarrow 9 \{ print(“9”) \}$

The parse tree for expression: 9-5+2



Using the depth first traversal of this tree, equivalent postfix expression : 95-2+

Writhing the Translation Schemes

Start with syntax directed definitions. Make sure that we never refer to an attribute that has not been defined already.

For s-attributed definitions, we can simply construct translation schemes by creating an action consisting of an assignment for each semantic rules into { ... } at the right most of each production

For example:

Production Semantic Rules

$S \rightarrow A_1 A_2$ $S.s = A_1.s + A_2.s$

$S \rightarrow a$ $A.s = 1$

The translation scheme is:

$S \rightarrow A_1 A_2 \{ S.s = A_1.s + A_2.s \}$

$S \rightarrow a \{ A.s = 1 \}$

If both synthesized and inherited attributes are involved,

- An inherited attribute for a symbol on the RHS of a production must be computed in an action before that symbol.
- An action must not refer to a synthesized attribute of a symbol that is to the right.
- A synthesized attribute of the NT on the LHS can only be computed after all attributes it references are already computed. (The actions for such attributes is placed in the right most end of production)

For example,

1. Incorrect

$S \rightarrow A_1 A_2 \{ A_1.in = 1; A_2.in = 2 \}$

$$S \rightarrow a\{print(A.in)\}$$

2. correct

$$S \rightarrow \{A_1.in = 1; A_2.in = 2\} A_1 A_2$$

$$S \rightarrow a\{print(A.in)\}$$

3. correct

$$S \rightarrow \{A_1.in = 1\} A_1 \{ A_2.in = 2 \} A_2$$

$$S \rightarrow a\{print(A.in)\}$$

Top down Translation:

Inherited attributes can be evaluated in a top-down fashion in a way similar to the technique used for elimination of left recursion. Consider the left recursive grammar with translation schemes.

$$A \rightarrow A_1 Y \{ A.a = g(A_1.a, Y.y) \} \quad (\text{here } A \text{ and } A_1 \text{ are same symbol})$$

$$A \rightarrow X \{ A.a = f(X.x) \}$$

Here, each attribute is synthesized. Eliminating the left recursion from above grammar the equivalent grammar is,

$$A \rightarrow XR$$

$$R \rightarrow YR \mid \epsilon$$

Taking the semantic actions into transformed grammar,

$$A \rightarrow X \{ R.i = f(X.x) \} R \{ A.a = R.s \}$$

$$R \rightarrow Y \{ R_1.i = g(R.i, Y.y) \} R_1 \{ R.s = R_1.s \}$$

$$R \rightarrow \epsilon \{ R.s = R.i \}$$

To see why the results of left recursive and non-recursive attributes are same, consider the following two annotated parse tree

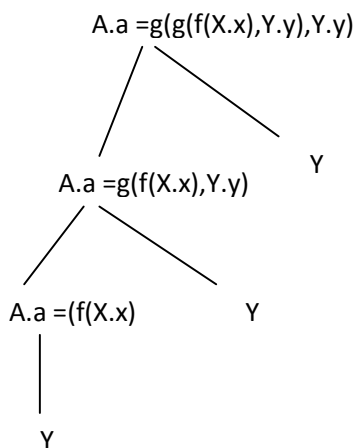


Figure 1

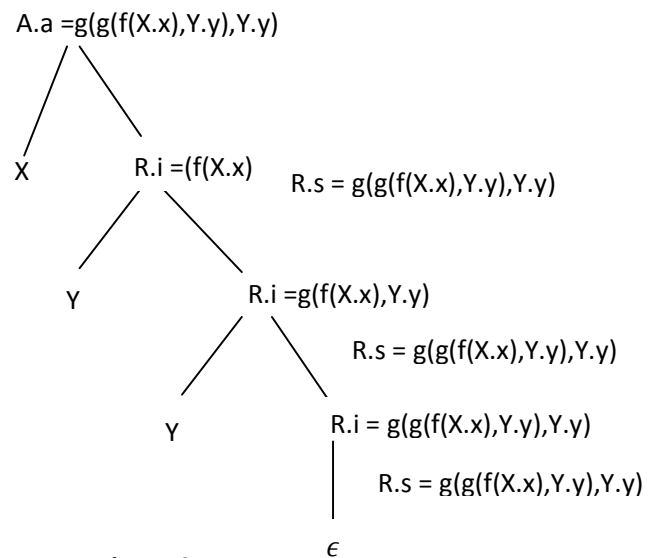


Figure 2

A.a computed according to above two translation scheme is same.

In figure 1, A.a is computed bottom up and in figure 2, R.i is computed as top down and A.a is computed by the R.i at the bottom is passed up unchanged as R.s.

Bottom up evaluation of L-attributed definitions

While evaluating S-attributed definitions during bottom-up parsing is straight forward, evaluating inherited attributed attributes is a bit tricky. L-attributed definitions are a simple subclass of inherited attributes that can be effectively implemented in most bottom up parsers.

Consider the type declaration of variables in a language like C described by the following rules.

$$\begin{aligned} D &\rightarrow T\{L.type = T.val\}L \\ L &\rightarrow L, id \{settype(id.entry, L.type)\} \\ L &\rightarrow id \{settype(id.entry, L.type)\} \\ T &\rightarrow int \{T.val = int\} \\ T &\rightarrow float \{T.val = float\} \end{aligned}$$

Given the string of the form **float id,id** , a bottom up evaluation can be traced as

STACK	INPUTBUFFER	Production Used	Action
\$	float id,id\$		shift
\$float	id,id\$	$T \rightarrow float$	Reduce
\$T {T.val =float}	id,id\$		shift
\$T id	,id\$	$L \rightarrow id$	reduce
\$T L{ L.type = T.val}	, id\$		shift
\$T L ,	id\$		shift
\$T L , id	\$	$L \rightarrow L , id$	Reduce
\$TL		$D \rightarrow TL$	reduce

The move of a parser for the given string from the grammar and the inherited attributes in parser stack.

For the above parser stack, the stack can be implemented as a pair of stacks(parallel stack) for state and value. The state stack is for the grammar symbol X i.e. $state[i]$ and value stack is used for holding the attributes of symbol on state stack. Every time the right side of the production for L is reduced in above example, T is in the stack just below the right side. We can use this fact to access the attribute value $T.type$ for evaluating the attributes.

If top and $ntop$ be the indices of the top entry in the stack just before and after a reduction takes place then from the copy rules defining $L.in = T.type$, $T.type$ is placed in $L.in$. So when $L \rightarrow id$ is applied for reduction, $val[top] = val[top-1]$. Similarly, since $L \rightarrow L , id$ is applied for reduction, there are 3 symbols on the right side of production so the 3 symbols in to top of state stack are removed and new symbol L is pushed to it at the same time the attribute for it in value stack will

be inherited as $\text{val}[\text{top}] = \text{val}[\text{top}-3]$. So the parser should execute some code fragments to obtain the attribute of symbol when reduction is applied. Following are the code fragments to be executed when the reduction is used.

Production	code fragments
$D \rightarrow TL$	
$L \rightarrow L, id$	$\text{val}[\text{top}] = \text{val}[\text{top}-3]$
$L \rightarrow id$	$\text{val}[\text{top}] = \text{val}[\text{top}-1]$
$T \rightarrow \text{int}$	$\text{val}[\text{ntop}] = \text{integer}$
$T \rightarrow \text{float}$	$\text{val}[\text{ntop}] = \text{float}$

Evaluation order of the attributes:

In syntax directed translation, the order of evaluation of the attributes is implemented by using the dependency graph. Once the dependency graph of the parse tree for a language construct is created for the given input string, the topological ordering the node of the dependency graph is the order of evaluation of attributes. If $b = f(x, y, z)$ is the semantic rule for any production of a grammar where attribute b depends on attributes x, y and z then x, y and z must be evaluated before b .

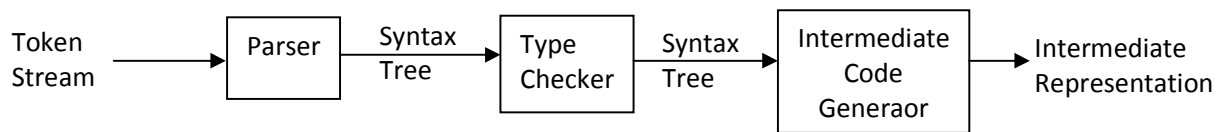
Chapter 6: Type Checking

Figure: Position of Type Checker

A compiler has to do semantic checks in addition to syntax analysis. Type checking is one of the static semantic checks but some systems also use dynamic type checking.

Static checking: the compiler enforces programming language's *static semantics*

- Program properties that can be checked at compile time

Static checking examples:

✓ Type checks.

- Report an error if an operator is applied to an incompatible operand e.g

```

int op(int), op(float);
int f(float);
int a, c[10], d;
d = c+d;           // FAIL type mismatch
*d = a;           // FAIL not a pointer type
a = op(d);         // OK: overloading (C++)
a = f(d);         // OK: coercion of d to float
vector<int> v;      // OK: template instantiation
  
```

✓ Flow-of-control checks.

- Statements that causes flow of control to leave a construct must have some
- place to which to transfer the flow of control e.g.

myfunc(int a) { cout<<a; break; // ERROR //missplaced break statement }	myfunc() { ... switch (a) { case 0: ... break; // OK case 1: ... }	myfunc(int a) { while(a) { ... if(i>10) Break; //ok } }
--	---	---

✓ Uniqueness checks.

- There are situations where an object must be defined exactly once.
- labels, identifiers e.g.

```

myfunc()
{ int i, j, i; // ERROR
...
}
  
```

✓ Name-related checks.

- Sometimes the same names may be appeared two or more times.
- Beginning and end of a construct

Dynamic semantics: checked at run time

- ✓ Compiler generates verification code to enforce programming language's dynamic semantics

A *type system* is a collection of rules for assigning type expressions to the parts of a program. A type checker implements a type system. A sound type system eliminates run-time type checking for type errors.

A programming language is strongly typed if its compiler can guarantee that there will be no type errors during run-time.

Type Expression:

The type of a language construct is defined by a “type expression”. A type expression is defined as follows:

- A Basic type is a type expression e.g. integer, char, real etc.
- A type name is a type expression e.g. if `int` is named by a variable `x`, then `x` is a type expression.
- A type constructor applied to a type expression is a type expression. The constructor include - array, product, pointer, function or record. e.g.
 - ✓ `T array[I]` or `array [I,T]` is a type expression with `I` elements of type `T`.
 - ✓ If `T1` and `T2` are type expressions, then Cartesian product `T1 X T2` is type expression - product
 - ✓ If `T` is a type expression, then `pointer(T)` is a type expression.
 - ✓ Function in programming languages is a mapping a domain type `D` to a range type `R`. e.g. `int x int → pointer(char)` denotes a function that takes a pair of integer and returns a pointer to char.
 - ✓ A record is a structured type

Specification of a simple type checker: The simple type checker is specified by the translation scheme that saves the type information for any identifier. Following is the translation scheme for a declaration.

$$\begin{aligned}
 P &\rightarrow D;E \\
 D &\rightarrow D;D \\
 D &\rightarrow id:T \quad \{ \text{addtype}(id.entry, T.val) \} \\
 T &\rightarrow char \quad \{ T.val = char \} \\
 T &\rightarrow int \quad \{ T.val = int \} \\
 T &\rightarrow real \quad \{ T.val = real \} \\
 T &\rightarrow *T_1 \quad \{ T.val = pointer(T_1.val) \} \\
 T &\rightarrow array[intnum] \text{ of } T_1 \quad \{ T.val = array(1..intnum.val, T_1.val) \}
 \end{aligned}$$

The above set of declarations describes a translation schemes that saves the type of an identifier in a language like pascal e.g. **a:integer** saves the type of id ‘a’ as ‘integer’

The type checking expression:

Following example shows the type checking expression in translation schemes

$$\begin{aligned}
 E \rightarrow id & \quad \{ E.type = lookup(id.entry) \} \\
 E \rightarrow literal & \quad \{ E.type = char \} \\
 E \rightarrow intliteral & \quad \{ E.type = int \} \\
 E \rightarrow realliteral & \quad \{ E.type = real \} \\
 E \rightarrow E_1 + E_2 & \quad \{ \text{if } (E_1.type = int \text{ and } E_2.type = int) \text{ then } E.type = int \\
 & \quad \quad \text{else if } (E_1.type = int \text{ and } E_2.type = real) \text{ then } E.type = real \\
 & \quad \quad \text{else if } (E_1.type = real \text{ and } E_2.type = int) \text{ then } E.type = real \\
 & \quad \quad \text{else if } (E_1.type = real \text{ and } E_2.type = real) \text{ then } E.type = real \\
 & \quad \quad \text{else } E.type = type-error \} \\
 E \rightarrow E_1 [E_2] & \quad \{ \text{if } (E_2.type = int \text{ and } E_1.type = array(s,t)) \text{ then } E.type = t \\
 & \quad \quad \text{else } E.type = type-error \} \\
 E \rightarrow *E_1 & \quad \{ \text{if } (E_1.type = pointer(t)) \text{ then } E.type = t \\
 & \quad \quad \text{else } E.type = type-error \}
 \end{aligned}$$
Another example: A simple language type checking specification:

$$\begin{aligned}
 E \rightarrow true & \quad \{ E.type = boolean \} \\
 E \rightarrow false & \quad \{ E.type = boolean \} \\
 E \rightarrow literal & \quad \{ E.type = char \} \\
 E \rightarrow num & \quad \{ E.type = integer \} \\
 E \rightarrow id & \quad \{ E.type = lookup(id.entry) \} \\
 E \rightarrow E_1 + E_2 & \quad \{ E.type := \text{if } E_1.type = integer \text{ and } E_2.type = integer \\
 & \quad \quad \text{then integer else type_error} \} \\
 E \rightarrow E_1 \text{ and } E_2 & \quad \{ E.type := \text{if } E_1.type = boolean \text{ and } E_2.type = boolean \\
 & \quad \quad \text{then boolean else type_error} \}
 \end{aligned}$$
Type Checking expression for another grammar: Example

$$\begin{aligned}
 T \rightarrow int & \quad \{ T.type = int \} \\
 T \rightarrow char & \quad \{ T.type = char \} \\
 T \rightarrow real & \quad \{ T.type = real \} \\
 T \rightarrow array [intnum, T_1] & \quad \{ T.type = array(1..intnum.val, T_1.type) \} \\
 T \rightarrow Pointer(T_1) & \quad \{ T.type = pointer(T_1.type) \}
 \end{aligned}$$
Type Checking expression of an array of pointers to real, where array index ranges from 1 to 100

$$\begin{aligned}
 T \rightarrow real & \quad \{ T.type = real \} \\
 E \rightarrow array [100, T] & \quad \{ \text{if } T.type = real \text{ then } T.type = array(1..100, T) \text{ else type_error}() \} \\
 E \rightarrow Pointer[E_1] & \quad \{ \text{if } (E_1.type = array[100, real]) \text{ then } E.type = E_1.type \text{ else } E.type = type-error \}
 \end{aligned}$$
Type checking expression of statements.**Assignment statement:**

$$S \rightarrow id = E \quad \{ \text{if } (id.type = E.type \text{ then } S.type = void \text{ else } S.type = type-error) \}$$
If then else statement:

$$S \rightarrow \text{if } E \text{ then } S_1 \quad \{ \text{if } (E.type = boolean \text{ then } S.type = S_1.type \text{ else } S.type = type-error) \}$$

While statement:

$$S \rightarrow \text{while } E \text{ do } S_1 \quad \{ \text{if } (E.\text{type}=\text{boolean} \text{ then } S.\text{type}=S_1.\text{type} \quad \text{else } S.\text{type}=\text{type-error} \}$$
Type Checking expression for function

$$E \rightarrow E_1 (E_2) \quad \{ \text{if } (E_2.\text{type}=s \text{ and } E_1.\text{type}=s \rightarrow t) \text{ then } E.\text{type}=t \\ \text{else } E.\text{type}=\text{type-error} \}$$

Ex: $\text{int } f(\text{double } x, \text{char } y) \{ \dots \}$
 $f: \quad \text{double } x \text{ char } \rightarrow \text{int}$
 argument types return type

Function whose domains are function from two characters and whose range is a pointer of integer.

$$\begin{aligned} T &\rightarrow \text{int} \{ T.\text{type} = \text{int} \} \\ T &\rightarrow \text{char} \{ T.\text{type} = \text{char} \} \\ T &\rightarrow \text{Pointer}[T_1] \{ T.\text{type} = \text{Pointer}(T_1.\text{type}) \} \\ E &\rightarrow E_1[E_2] \{ \text{if } (E_2.\text{type} = (\text{char}, \text{char}) \text{ and } E_1.\text{type} = (\text{char}, \text{char}) \rightarrow \text{Pointer}(\text{int}) \\ &\quad \text{then } E.\text{type} = E_1.\text{type} \text{ else } \text{type_error} \} \end{aligned}$$

Example: consider the following grammar for arithmetic expression using an operator 'op' to integer or real numbers

$$E \rightarrow E_1 \text{op } E_2 \mid \text{num.num} \mid \text{num} \mid \text{id}$$

Give the syntax directed definition as translation scheme to determine the type of expression when two integers are used in expression resulting type is int otherwise real

Translation scheme will be:

$$\begin{aligned} E &\rightarrow \text{id} \quad \{ E.\text{type}=\text{lookup}(\text{id.entry}) \} \\ E &\rightarrow \text{num} \quad \{ E.\text{type}=\text{integer} \} \\ E &\rightarrow \text{num.num} \quad \{ E.\text{type}=\text{real} \} \\ E &\rightarrow E_1 \text{op } E_2 \quad \{ \text{if } (E_1.\text{type}=\text{integer} \text{ and } E_2.\text{type}=\text{integer}) \text{ then } E.\text{type}=\text{integer} \\ &\quad \text{else if } (E_1.\text{type}=\text{integer} \text{ and } E_2.\text{type}=\text{real}) \text{ then } E.\text{type}=\text{real} \\ &\quad \text{else if } (E_1.\text{type}=\text{real} \text{ and } E_2.\text{type}=\text{integer}) \text{ then } E.\text{type}=\text{real} \\ &\quad \text{else if } (E_1.\text{type}=\text{real} \text{ and } E_2.\text{type}=\text{real}) \text{ then } E.\text{type}=\text{real} \\ &\quad \text{else } E.\text{type}=\text{type-error} \} \end{aligned}$$
Type Conversion and Coercion

Type conversion is explicit, for example using type casts. Consider expression $a + b$ where a is of type integer and b is real. Since the representation of integer and real in the computer system is different and different machine instructions are used for operations of integer and reals, the compiler must convert one type operand into another type operand of operator $+$ to ensure both operands are of same type when addition is takes place.

The conversion of type of one operand to another can be explicitly using cast operators that the type checker must incorporate.

Type coercion is implicitly performed by the compiler to generate code that converts types of values at runtime (typically to *narrow* or *widen* a type)

Both require a *type system* to check and infer types from (sub)expressions

Equivalence of Type expression:

As long as type expressions are built from basic types and constructors, a notion of equivalence between two type expressions is structural equivalence. Two type expressions are structurally equivalent iff they are identical. E.g. `pointer(Integer)` is equivalent to `pointer(Integer)`.

The algorithm for testing structural equivalence can be adapted to test the equivalence. If two type expressions are equal then the operation is performed otherwise it requires type conversions if possible or report type error. For example, if two operand are of int and real type, then operation can be performed by type checking and conversion but if an array and a record is operated as `a+r`, then there will be type error.

Following is the example of an algorithm that can be adopted to check the equivalence of the type expression.

```
boolean sequival (s, t)
{
    if s and t are the same basic type
        return true;
    else if s=array(s1,s2) and t=array(t1,t2) then
        return sequival(s1,t1) and sequival(s2,t2)
    else if s=s1 x s2 and t=t1 x t2 then
        return sequival(s1,t1) and sequival(s2,t2)
    else if s= pointer(s1) and t=pointer(t1) then
        return sequival(s1,t1)
    else if s=s1->s2 and t=t1->t2 then
        return sequival(s1,t1) and sequival(s2,t2)
    else return false
}
```

If array bound `s1` and `t1` in `s=array(s1,s2)` and `t=array(t1,t2)` are ignored if the test for array equivalence in above then it can be re-formulated as:

```
if s=array(s1,s2) and t=array(t1,t2) then
    return sequival(s2,t2)
```

Intermediate Code Generation

The given program in a source language is converted into an equivalent program in an intermediate language by the intermediate code generator. Intermediate codes are machine independent codes. The intermediate representation are :

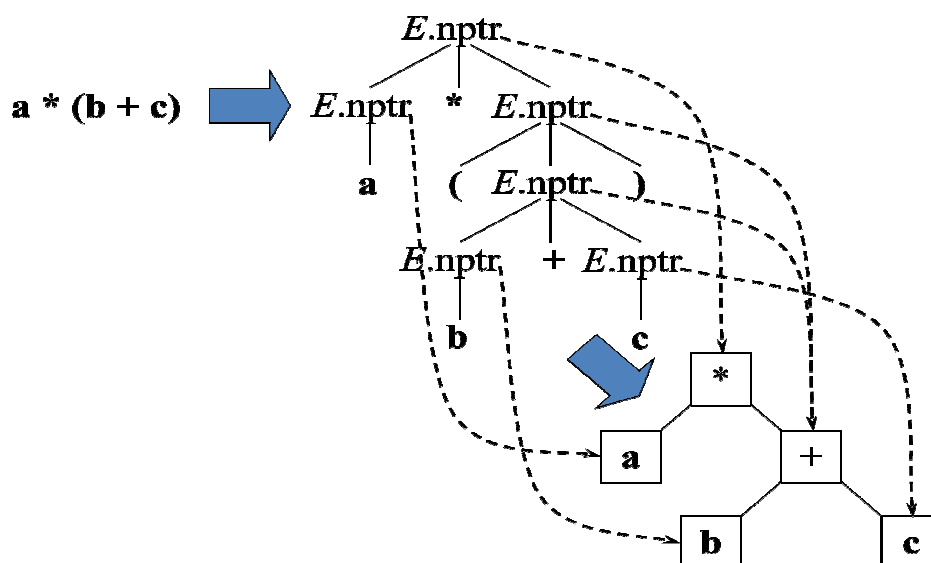
- Graphical representation e.g. Abstract Syntax Tree(AST) , DAGS
- Postfix notations
- Three Address codes

Some languages have well defined intermediate codes e. g. JAVA – JVM

Let us take an example of a syntax directed definitions for assignment for intermediate representation (AST)

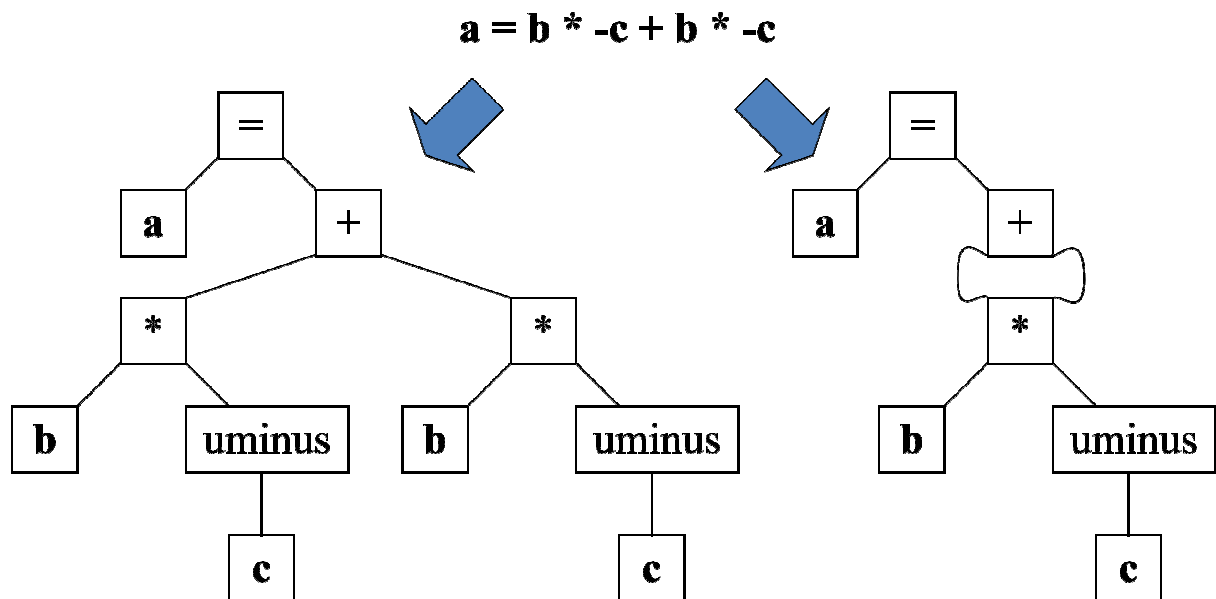
Production	SemanticRule
$S \rightarrow \text{id} = E$	$S.\text{nptr} = \text{mknode}(':=', \text{mkleaf}(\text{id}, \text{id.entry}), E.\text{nptr})$
$E \rightarrow E_1 + E_2$	$E.\text{nptr} = \text{mknode}('+', E_1.\text{nptr}, E_2.\text{nptr})$
$E \rightarrow E_1 * E_2$	$E.\text{nptr} = \text{mknode}('*', E_1.\text{nptr}, E_2.\text{nptr})$
$E \rightarrow -E_1$	$E.\text{nptr} = \text{mknode}('uminus', E_1.\text{nptr})$
$E \rightarrow (E_1)$	$E.\text{nptr} := E_1.\text{nptr}$
$E \rightarrow \text{id}$	$E.\text{nptr} := \text{mkleaf}(\text{id}, \text{id.entry})$

The Abstract syntax tree for the expression $a * (b + c)$ can be as



Compiler Design and Construction

Abstract syntax tree vs DAGS for expression : $a = b * -c + b * -c$



Three Address Code:

A three address code is a sequence of the general form $x = y \text{ op } z$, where x, y and z are names, constants or compiler generated temporaries and op is for operator. So the source language expression like $x + y * z$ might be translated into a sequence as

$t1 = y * z$

$t2 = x + t1$

where $t1$ and $t2$ are compiler generated temporaries.

Similarly, $x = y + z * w$ should be represented as

$t1 = z * w$

$t2 = y + t1$

$x = t2$ et.

So code for $a = b * -c + b * -c$ should be,

For AST:

$t1 = -c$

$t2 = b * t1$

$t3 = -c$

$t4 = b * t1$

$t5 = t2 + t4$

$a = t5$

For DAG:

$t1 = -c$

$t2 = b * t1$

$t3 = t2 + t2$

$a = t3$

Compiler Design and Construction

In fact three address code is linearization of tree. Followings are the 3-address code statements for different statements.

Assignment statement: $x = y \text{ op } z$ (binary operator op)

$x = \text{op } y$ (unary operator op)

Copy statement: $x = z$ (also called copy assignment)

Unconditional jump: goto L (jump to label L)

Conditional jump : if x relop y goto L

Procedure call:

param x1

param x2

.

.

param x_n

call p,n i.e. call procedure $P(x_1, x_2, \dots, x_n)$

Indexed assignment:

$x = y[i]$

$x[i] = y$

Address and pointer assignments:

$x = \&y$

$x = *y$

$*x = y$

When three address code is generated temporary names are made up for the interior nodes of a syntax tree. For production $E \rightarrow E_1 + E_2$, the value of non terminal E is computed into a new temporary t.

Syntax Directed Translation into 3-address codes

- First deal with the assignment
- Use attributes
 - E.place : the name that will hold the value of E
 - E.code : hold the three address code statements that evaluates E
- Use function newtemp() that returns a new temporary variable that we can use.

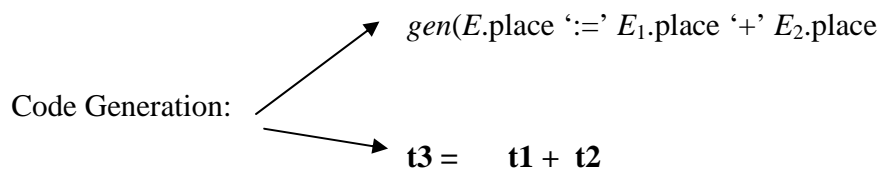
Compiler Design and Construction

- Use a function say `gen()` to generate a single three-address statement given the necessary information. e.g. `gen(E.place ':=' E1.place '+' E2.place)`
 - $\Rightarrow \mathbf{t3 = t1 + t2}$

For example:

$S \rightarrow \mathbf{id} = E$	$\{ S.code = E.code \parallel gen(\mathbf{id.place} \text{ '=' } E.place); \}$
$E \rightarrow E_1 + E_2$	$\{ E.place = newtemp(); E.code = E_1.code \parallel E_2.code \parallel gen(E.place \text{ '=' } E_1.place \text{ '+' } E_2.place) \}$
$E \rightarrow E_1 * E_2$	$\{ E.place = newtemp();$ $E.code = E_1.code \parallel E_2.code \parallel gen(E.place \text{ '=' } E_1.place \text{ '*' } E_2.place) \}$
$E \rightarrow -E_1$	$\{ E.place = newtemp();$ $E.code = E_1.code \parallel gen(E.place \text{ '=' 'uminus' } E_1.place) \}$
$E \rightarrow (E_1)$	$\{ E.place = E_1.place$ $E.code := ' ' \}$
$E \rightarrow \mathbf{id}$	$\{ E.place = \mathbf{id.entry}$ $E.code := ' ' \}$
$E \rightarrow \mathbf{num}$	$\{ E.place = newtemp();$ $E.code := gen(E.place \text{ '=' } \mathbf{num.value}) \}$

Here;



Implementation of Three address codes:

A three address statement is an abstract form of intermediate code. In compiler such statements can be implemented as records with fields for the operators and operands

Followings are the three address statements representations.

1. Quadruples:
 - A records structure with four fields: op, arg1, arg2, result.
 - The three address statements $x = y \text{ op } z$ is represented by placing y in arg1, z in arg2 and x in result.

Compiler Design and Construction

- statement with unary operators like $x = -y$ or $x = y$ do not use arg2.
- operators like param use neither arg2 nor result
- conditional and unconditional jumps put the target label in result.

e.g. for statement $a = b * -c + b * -c$, the quadruples is:

pos	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

2. Triples:

pos	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

3. Indirect triples:

- Indirect triples involve in listing of pointers to triples rather than listing of triples themselves. e.g. the three address indirect triples for same statement above is

pos	op		pos	op	arg1	arg2
(0)	(10)	→	(10)	uminus	c	
(1)	(11)	→	(11)	*	b	(10)
(2)	(12)	→	(12)	uminus	c	
(3)	(13)	→	(13)	*	b	(12)
(4)	(14)	→	(14)	+	t2	(13)
(5)	(15)	→	(15)	=	t5	(14)

Assignments and Symbol Table

We assume that names/ addresses stand for pointer to their symbol table entries since other info are needed for final code generation.

Compiler Design and Construction

- Under this assumption, temporary names must be also entered into symbol table as they are created by the newtemp function.
- The lexeme for name id is given by id.entry and the function lookup(id.entry) returns null if there is no entry found otherwise a ptr to the entry is returned.
- Instead of using code attribute, let a procedure emit() produce three address code to output file.
 - This is always possible if the code of the non-terminal on the left is obtained by concatenating the code attributes of the non-terminals on the right in the same order. e.g. $E \rightarrow E_1 \text{ op } E_2$ { E.place = newtemp

emit(E.place '=' E1.place 'op' E2.place)

using this approach, the translation scheme to produce the three address code for assignment can be written as:

$S \rightarrow \text{id} = E$	<i>{ p = lookup(id.entry);</i>
	<i>if p!= null then emit(p '=' E.place) else error;</i>
$E \rightarrow E_1 + E_2$	<i>{ E.place=newtemp();</i>
	<i>emit(E.place '=' E₁.place '+' E₂.place) }</i>
$E \rightarrow E_1 * E_2$	<i>{ E.place=newtemp();</i>
	<i>emit(E.place '=' E₁.place '*' E₂.place) }</i>
$E \rightarrow -E_1$	<i>{ E.place=newtemp();</i>
	<i>emit(E.place '=' 'uminus' E₁.place)}</i>
$E \rightarrow (E_1)$	<i>{ E.place=E₁.place }</i>
$E \rightarrow \text{id}$	<i>{ p=lookup(id.entry)</i>
	<i>if p!=NULL then E.place=p else error</i>
	<i>}</i>

Boolean Expression:

Boolean expressions are used to either compute logical values or as conditional expressions in flow of control statements

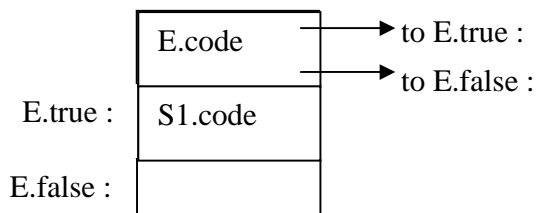
- We considered Boolean expression with the following grammar
 1. $E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid \text{id rel op id} \mid \text{true} \mid \text{false}$
- There are two methods to evaluate Boolean expressions
 1. Numerical Representation : Encode true with '1' and false with '0' and we proceed analogously to arithmetic expression.
 2. Jumping Code: We represent the value of a Boolean expression by a position reached in a program.

- Jumping code is extremely useful when Boolean expression are in the context of flow of control statement.
- consider the follow of control statements generated by following grammar.

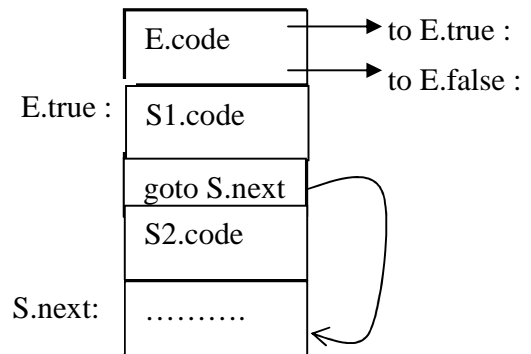
$S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S1 \text{ else } S2 \mid \text{while } E \text{ do } S$

Flow of control statements:

- In the translation, we assume that a three address code statement can have a **symbolic label** and that the function **newlabel** generates such labels:
- We associate with E to labels using inherited attributes:
 1. E.true – the label to which control flows if E is true.
 2. E.false - the label to which control flows if E is false.
- We associate to S the inherited attribute S.next that represents the label attached to the first statement after the code for S.
- The following figures show how the flow of control statements are translated.



if -then:



if –then-else

Procedure Calls: The procedure call can be represented in three address code as below. Let us consider the following grammar.

$S \rightarrow \text{call id}(\text{Elist})$

$\text{Elist} \rightarrow \text{Elist} , E \mid E$

The procedure call statement : **call fun(a+1, b, 7)** should be represented as:

t1=a+1

t2=7

param t1

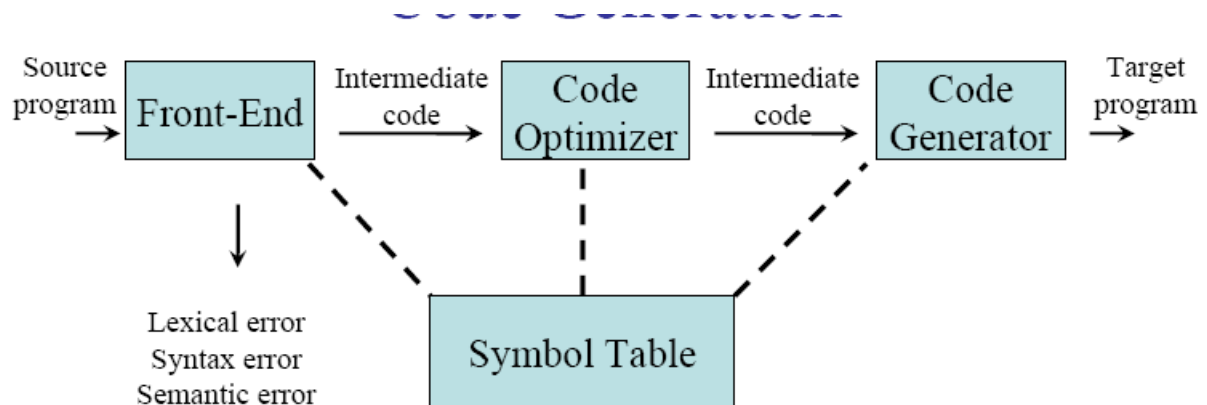
param b

param t2

call fun 3

Code Generation & Optimization

How the target codes are generated optimally from an intermediate form of programming language.



Code produced by compiler must be correct and high quality. Source-to-target program transformation should be *semantics preserving* and effective use of target machine resources. Heuristic techniques should be used to generate good but suboptimal code, because generating optimal code is un-decidable.

Code Generator Design Issues

The details of code generation are dependent on the target language and operating system. Issues such as memory management, instruction selection, register allocation, evaluation order are in almost all code – generation problems.

The issues to code generator design includes:

Input to the code generator: The input to the code generator is intermediate representation together with the information in the symbol table. What type of input postfix, three-address, dag or tree.

Target Program: Which one is the out put of code generator: Absolute machine code (executable code), Re-locatable machine code (object files for linker), Assembly language (facilitates debugging), Byte code forms for interpreters (e.g. JVM)

Target Machine: Implementing code generation requires thorough understanding of the target machine architecture and its instruction set.

Instruction Selection: Efficient and low cost instruction selection is important to obtain efficient code.

Register Allocation: Proper utilization of registers improve code efficiency

Choice of Evaluation order: The order of computation effect the efficiency of target code.

The Target Machine

Consider a hypothetical target computer is a byte-addressable machine (word = 4 bytes) and n general propose registers, R0, R1, ..., Rn-1. It has two address instruction of the form:

op source, destination

It has the following op-codes :

MOV (move content of source to destination),

ADD (add content of source to destination)

SUB (subtract content of source from destination .)
MUL (multiply content of source with destination)

The source and destination of instructions are specified by combining register and memory location with address modes. The address mode together with assembly forms and associated cost are:

Addressing modes:

Mode	Form	Address	Added Cost
Absolute	M	M	1
Register	R	R	0
Indexed	$c(\mathbf{R})$	$c + \text{contents}(\mathbf{R})$	1
Indirect register	*R	$\text{contents}(\mathbf{R})$	0
Indirect indexed	$\mathbf{*}c(\mathbf{R})$	$\text{contents}(c + \text{contents}(\mathbf{R}))$	1
Literal	#c	N/A	1

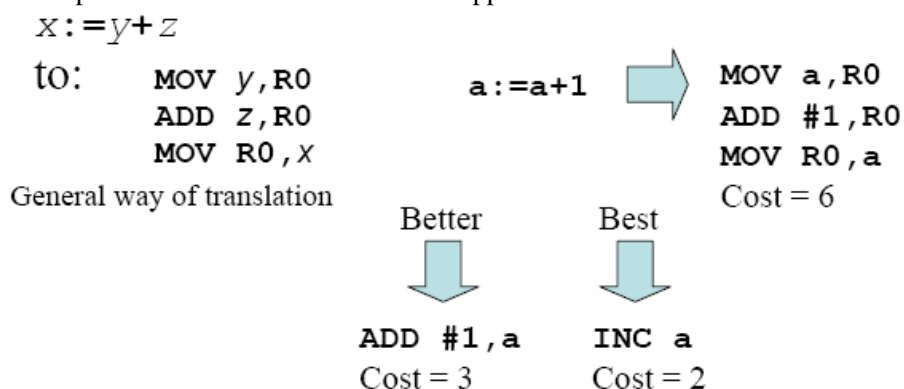
Instruction Costs

- Machine is a simple, non-super-scalar processor with fixed instruction costs
- Realistic machines have deep pipelines, I-cache, D-cache, etc.
- Define the cost of instruction

Instruction	operation
MOV R0,R1	Store $\text{content}(\mathbf{R0})$ into register R1 1
MOV R0,M	Store $\text{content}(\mathbf{R0})$ into memory location M 2
MOV M,R0	Store $\text{content}(\mathbf{M})$ into register R0 2
MOV 4(R0),M	Store $\text{contents}(4 + \text{contents}(\mathbf{R0}))$ into M 3
MOV *4(R0),M	Store $\text{contents}(\text{contents}(4 + \text{contents}(\mathbf{R0})))$ into M 3
MOV #1,R0	Store 1 into R0 2
ADD 4(R0),*12(R1)	Add $\text{contents}(4 + \text{contents}(\mathbf{R0}))$ to value at location $\text{contents}(12 + \text{contents}(\mathbf{R1}))$ 3

Instruction Selection

Instruction selection is important to obtain efficient code. Suppose we translate three-address code



Picking the shortest sequence of instructions is often a good approximation of the optimal result

Register Allocation and Assignment

Accessing values in registers is much faster than accessing main memory. *Register allocation* denotes the selection of which variables will go into registers. *Register assignment* is the determination of exactly which register to place a given variable. The goal of these operations is generally to minimize the total number of memory accesses required by the program.

Finding an optimal register assignment in general is NP-complete.

Register Allocation and Assignment Example

```
t := a * b
t := t + a
t := t / d
```

↓ { R1 = t }

```
MOV a, R1
MUL b, R1
ADD a, R1
DIV d, R1
MOV R1, t
```

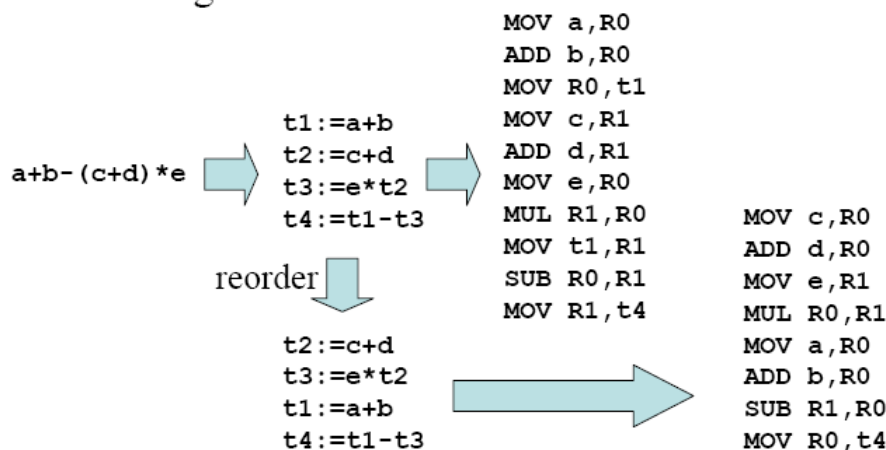
```
t := a * b
t := t + a
t := t / d
```

↓ { R0 = a, R1 = t }

```
MOV a, R0
MOV R0, R1
MUL b, R1
ADD R0, R1
DIV d, R1
MOV R1, t
```

Choice of Evaluation Order

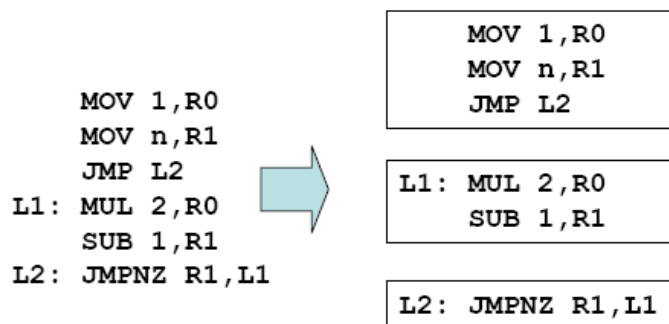
When instructions are independent, their evaluation order can be changed



Basic Blocks and Control Flow Graphs

Basic Blocks

A *basic block* is a sequence of consecutive instructions in which flow of control enters by one entry point and exit to another point without halt or branching except at the end.



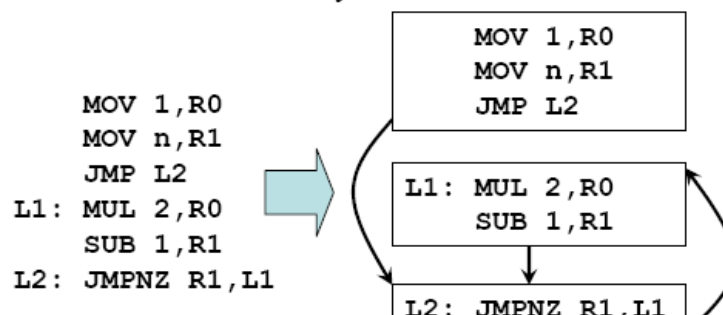
Basic Blocks and Control Flow Graphs

Flow Graphs

A *flow graph* is a graphical depiction of a sequence of instructions with control flow edges.

A flow graph can be defined at the intermediate code level or target code level.

The nodes of flow graphs are the basic blocks and flow-of-control to immediately follow node connected by directed arrow.



Basic Blocks Construction Algorithm

Input: A sequence of three-address statements

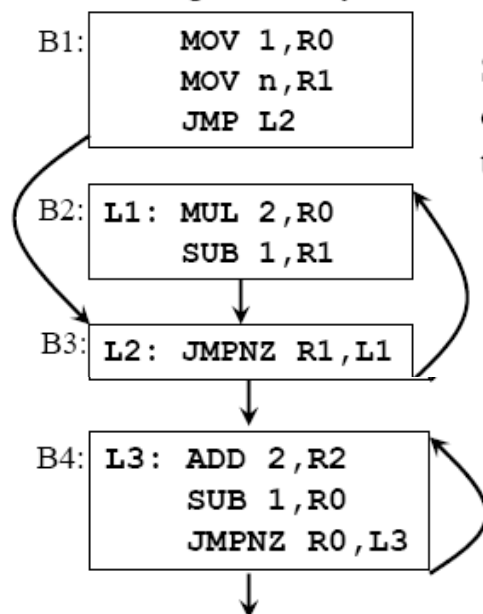
Output: A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements of basic blocks
 - a. The first statement is the leader
 - b. Any statement that is the target of a conditional or goto is a leader
 - c. Any statement that immediately follows conditional or goto is a leader
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program

Loops

A *loop* is a collection of basic blocks, such that

- All blocks in the collection are *strongly connected*
- The collection has a unique *entry*, and the only way to reach a block in the loop is through the entry



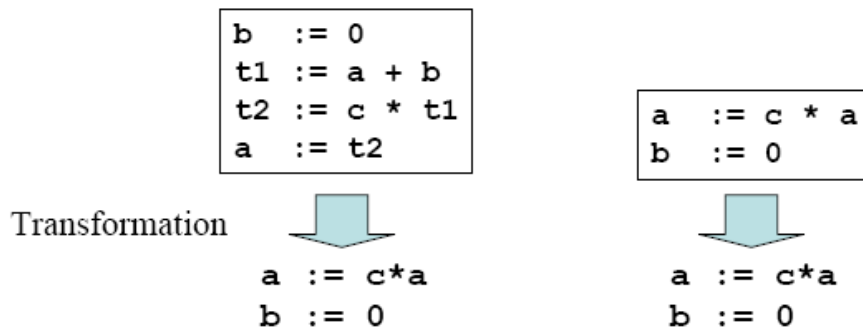
Strongly connected components: there is path of length of one or more from one node to another to make a cycle. Such as {B2,B3}, {B4}

Entries: B3, B4

A loop that consists no other loop is called inner loop

Equivalence of Basic Blocks

Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions



Blocks are equivalent, assuming `t1` and `t2` are *dead*: no longer used (no longer *live*)

Transformations on Basic Blocks

A *code-improving transformation* is a **code optimization** to improve speed or reduce code size

Global transformations are performed across basic blocks

Local transformations are only performed on single basic blocks

Transformations must be safe and preserve the meaning of the code

A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form

Some local transformation are:

- Common-Subexpression Elimination
- Dead Code Elimination
- Renaming Temporary Variables
- Interchange of Statements
- Algebraic Transformations

Common-Subexpression Elimination

Remove redundant computations

Look at 2nd and 4th:
compute same
expression

```
a := b + c
b := a - d
c := b + c
d := a - d
```



```
a := b + c
b := a - d
c := b + c
d := b
```

Look at 1st and 3rd:
b is redefine in 2nd
therefore different in
3rd, not the same
expression

```
t1 := b * c
t2 := a - t1
t3 := b * c
t4 := t2 + t3
```



```
t1 := b * c
t2 := a - t1
t4 := t2 + t1
```

Dead Code Elimination

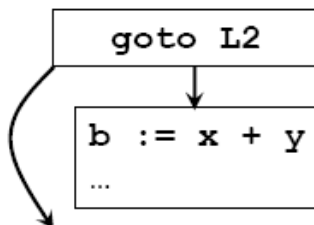
Remove unused statements

```
b := a + 1
a := b + c
...
```



```
b := a + 1
...
```

Assuming **a** is *dead* (not used)

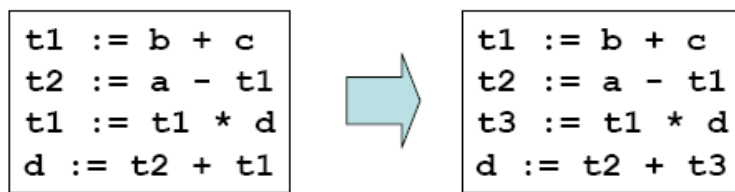


Remove unreachable code

Renaming Temporary Variables

Temporary variables that are dead at the end of a block can be safely renamed

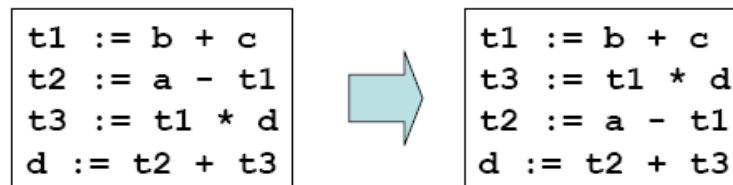
The basic block is transformed into an equivalent block in which each statement that defines a temporary defines a new temporary. Such a basic block is called *normal-form block* or *simple block*.



Normal-form block

Interchange of Statements

Independent statements can be reordered without effecting the value of block to make its optimal use.

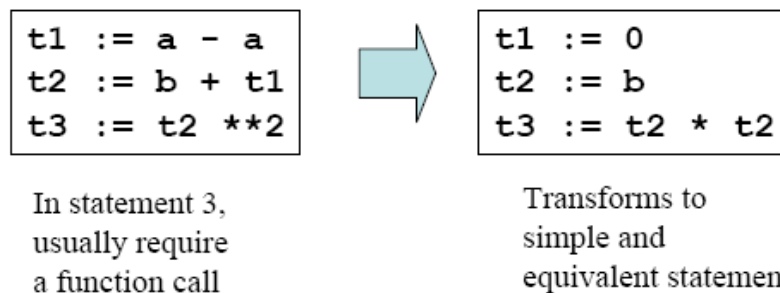


Note that normal-form blocks permit all statement interchanges that are possible

Algebraic Transformations

Change arithmetic operations to transform blocks to algebraic equivalent forms

Simplify expression or replace expensive expressions by cheaper ones.



Next-Use Information

Next-use information is needed for dead-code elimination and register assignment (if the name in a register is no longer needed, then the register can be assigned to some other name)

If $i: x = \dots$ and $j: y = x + z$ are two statements i & j , then *next-use* of x at i is j .

Next-use is computed by a backward scan of a basic block and performing the following actions on statement

$i: x := y \text{ op } z$

- Add liveness/next-use info on x , y , and z to statement i (whatever in the symbol table)

- Before going up to the previous statement (scan up):

- Set x info to “not live” and “no next use”
- Set y and z info to “live” and the next uses of y and z to i

All nontemporary variables and temporary that is used across the block are considered live.

Computing Next-Use

Example

Step 1

$i: a := b + c$

$j: t := a + b$ [$live(a) = true, live(b) = true, live(t) = true,$
 $nextuse(a) = none, nextuse(b) = none, nextuse(t) = none$]

Attach current live/next-use information

Because info is empty, assume variables are live

(Data flow analysis Ch.10 can provide accurate information)

Step 2

$i: a := b + c$

$live(a) = true$	$nextuse(a) = j$
$live(b) = true$	$nextuse(b) = j$
$live(t) = false$	$nextuse(t) = none$

$j: t := a + b$ [$live(a) = true, live(b) = true, live(t) = true,$
 $nextuse(a) = none, nextuse(b) = none, nextuse(t) = none$]

Compute live/next-use information at j

Computing Next-Use

Step 3 $i: a := b + c$ [$live(a) = true, live(b) = true, live(c) = false,$
 $nextuse(a) = j, nextuse(b) = j, nextuse(c) = none$]

$j: t := a + b$ [$live(a) = true, live(b) = true, live(t) = true,$
 $nextuse(a) = none, nextuse(b) = none, nextuse(t) = none$]

Attach current live/next-use information to i

Step 4

$live(a) = false$	$nextuse(a) = none$
$live(b) = true$	$nextuse(b) = i$
$live(c) = true$	$nextuse(c) = i$
$live(t) = false$	$nextuse(t) = none$

$i: a := b + c$ [$live(a) = true, live(b) = true, live(c) = false,$
 $nextuse(a) = j, nextuse(b) = j, nextuse(c) = none$]

$j: t := a + b$ [$live(a) = false, live(b) = false, live(t) = false,$
 $nextuse(a) = none, nextuse(b) = none, nextuse(t) = none$]

Compute live/next-use information i

Code Generator

Generates target code for a sequence of three-address statements using next-use information

Uses new function *getreg* to assign registers to variables

Computed results are kept in registers as long as possible, which means:

- Result is needed in another computation
- Register is kept up to a procedure call or end of block

Checks if operands to three-address code are available in registers

Code Generation Algorithm

For each statement $x := y \text{ op } z$

1. Set location $L = \text{getreg}(y, z)$ // to store the result of $y \text{ op } z$
2. If $y \notin L$ then generate //L is address descriptor --wait!
MOV y', L //to place copy of y in L
where y' denotes one of the locations where the value of y is available (choose register if possible)
3. Generate instruction
OP z', L
where z' is one of the locations of z ;
Update register/address descriptor of x to include L
4. If y and/or z has no next use and is stored in register, update register descriptors to remove y and/or z

Register and Address Descriptors

A *register descriptor* keeps track of what is currently stored in a register at a particular point in the code, e.g. a local variable, argument, global variable, etc.

MOV a, R0 “R0 contains a”

An *address descriptor* keeps track of the location where the current value of the name can be found at run time, e.g. a register, stack location, memory address, etc.

MOV a, R0

MOV R0, R1 “a in R0 and R1”

The *getreg* Algorithm

To compute *getreg*(*y*, *z*)

1. If *y* is stored in a register *R* and *R* only holds the value *y*, and *y* has no next use, then return *R*;
Update address descriptor: value *y* no longer in *R*
2. Else, return a new empty register if available
3. Else, find an occupied register *R*;
Store contents (register spill) by generating
MOV R, M
for every *M* in address descriptor of *y*;
Return register *R*
4. If not used in the block or no suitable register return a memory location

Code Generation

Example

Statement: $d := (a-b) + (a - c) + (a - c)$

Statements	Code Generated	Register Descriptor	Address Descriptor
$t := a - b$	MOV a,R0 SUB b,R0	Registers empty R0 contains t	t in R0
$u := a - c$	MOV a,R1 SUB c,R1	R0 contains t R1 contains u	t in R0 u in R1
$v := t + u$	ADD R1,R0	R0 contains v R1 contains u	u in R1 v in R0
$d := v + u$	ADD R1,R0 MOV R0,d	R0 contains d	d in R0 d in R0 and memory

Peephole Optimization

Statement-by-statement code generation often produce redundant instructions that can be optimize to save time and space requirement of target program.

Examines a short sequence of target instructions in a window (*peephole*) and replaces the instructions by a faster and/or shorter sequence whenever possible.

Applied to intermediate code or target code

Typical optimizations:

- Redundant instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

Eliminating Redundant Loads and Stores

Consider

```
MOV R0, a
MOV a, R0
```

This type code is not generated by our algorithm of page 25

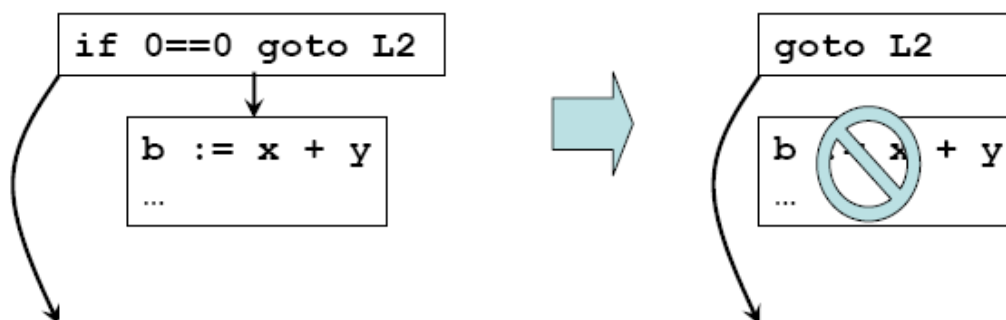
The second instruction can be deleted because first ensures value of `a` in `R0`, but only if it is not labeled with a target label

- Peephole represents sequence of instructions with at most one entry point

The first instruction can also be deleted if $live(a) = false$

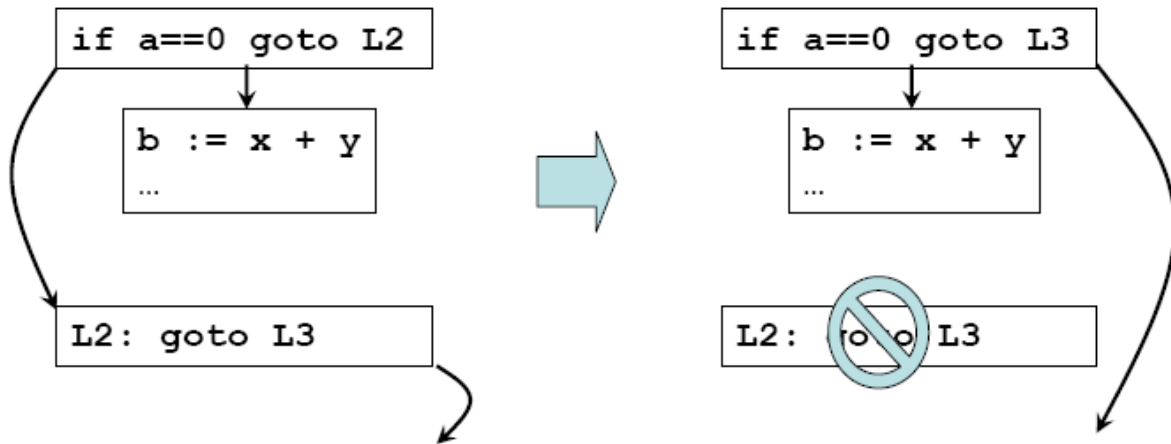
Deleting Unreachable Code

An unlabeled instruction immediately following an unconditional jump can be removed



Branch Chaining

Shorten chain of branches by modifying target labels



Remove redundant jumps as well

```
goto L1
...
L1: if a < b goto L2
    goto L3
.....
```

→

```
if a < b goto L2
goto L3
.....
```

Other Peephole Optimizations

Reduction in strength: replace expensive arithmetic operations with cheaper ones

```
...
a := x ^ 2
b := y / 8
```

→

```
...
a := x * x
b := y >> 3
```

Utilize machine idioms (use addressing mode inc)

```
...
a := a + 1
```

→

```
...
inc a
```

Algebraic simplifications

```
...
a := a + 0
b := b * 1
```

→

```
...
```

Run Time Storage management

A compiler contains a block of storage from the operating system for the compiled program to run in. This run time storage might be sub-divided to hold

1. The generated target code
 2. data objects and
 3. a counterpart of the control stack to keep track of procedure activation.
- The size of generated target code is fixed at compile time so it can be placed in a statically determined area – low end of memory.
 - Some of data objects may also be known at compile time so these too can be placed in to statically determined area.
 - The addresses of these data objects can be compiled into target code
 - For the activation of procedure, when a call occurs, execution of an activation is interrupted and information about the status of the machine such as value of program counter, machine register is saved into stack until the control returns from call to the activation.
 - Data objects whose life times are contained in that of an activation can be allocated on the stack along with other information associated with the activation.
 - Separate area of run time storage, called heap, holds other information.

The management of run time storage by sub-division is:

Code
Static Data
Stack
Heap

- The size of stack and heap may change during execution.
- By convention, stack grows down and heap grows up

Information needed by a single execution of a procedure is managed using a contiguous block of storage called an activation record:

An activation record is a collection of fields, starting from the field for temporaries as

Returned value	← value returned after execution
actual parameter	← used by the calling procedure to call procedure.
optional control link	← points to the activation record of the caller.
optional access link	← Non local data held in other activation record.
saved machine state	← State of the machine just before procedure call
local data	← Data that are local to an execution.
temporaries	← Temporary values used for evaluation of expression

Since, run time allocation and de-allocation of activation records occurs as part of procedure call-return sequences, following three address statements are in focus.

1. call
2. return
3. halt
4. action – a place holder for other statements

Now , consider the following input to the code generator.

Three address code

<pre>/*Code for procedure C */ action 1 call p action 2 halt</pre>
<pre>/*code for proceure p */ action 3 return</pre>

Activation record for C 64 bytes

0:	return address
	Array
56:	i
60:	j

Activation record for p(88 b)

0:	return address
4:	Buffer
84:	n

Using the static allocation,

A call statement in the intermediate code is implemented by two target machine instruction MOV and GOTO

- The code constructed from procedure C and p above using arbitrary address 100 and 200 as:

Assume action takes cost of 20 bytes. – MOV and GOTO + 3 constants cost = 20 bytes

The target code for the input above will be as:

100: ACTION1

120: MOV #140,364 /* saves return address 140 */

132: ACTION2

160: HLT

.....

/*Code for P */

200: ACTION3

220: GOTO *364 /* returns to address saved in location 364 */

.....

/* 300-363 hold activation record for c */

300: /* return address */

304: /*local data for c */

.....

/* 364-451 holds activation record for P */

364: /*return address */

368: /* local data for p */

- The MOV instruction at address 120 saves the return address 140 in machine status field - the first word in activation record of p.
- The GOTO instruction at 132 transfers control to first instruction to the target code of called procedure.
- *364 represents 140 when GOTO statement at address 220 is executed, control then returns to 140.